INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




CI/CD Pipeline para um Sistema Monolítico

RAFAEL FERNANDES FAÍSCA
Setembro de 2025

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

CI/CD Pipeline for a Monolithic System
A .NET Framework Case Study

Rafael Faísca

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Nuno Bettencourt

Porto, September 24, 2025

Declaration of Integrity

I hereby declare that I have conducted this academic work with the utmost integrity. I affirm that I have neither engaged in plagiarism nor misused information, nor falsified any results during the development and preparation of this work.

I confirm that the content presented in this document is original and entirely my own authorship and has not been previously submitted or used for any other purpose.

Furthermore, I acknowledge that I am fully aware of the ethical principles outlined in P.PORTO's Code of Ethical Conduct and have adhered to them throughout this academic endeavor.

ISEP, Porto, September 24, 2025

Rafael Fernandes Faísca

Dedicatory

I dedicate this work to my amazing teammates, whose cooperation, encouragement, and collective will throughout my master's program made the experience both rewarding and inspiring.

I want to sincerely thank my supervisor and my professors, whose invaluable advice, insight, and support have influenced this journey and greatly aided in my development.

Lastly, to my family, who have been my biggest source of strength because of their unwavering love, tolerance, and faith in me. We appreciate everyone's contributions to this significant occasion.

Abstract

This dissertation examines how a continuous integration and delivery (CI/CD) pipeline can be introduced—and fairly judged—in DOMUS, a large, multi-component platform developed at P.PORTO. DOMUS has grown fast and wide; without CI/CD, everyday work exposes underlying inefficiencies: versions drift, dependencies surprise people at build time, deployments take longer than they should, and coordination frays. The project therefore sets out to design a CI/CD process that fits DOMUS as it actually is, not as a neat diagram. Which we then evaluate against a CI/CD maturity model.

Alongside the engineering work, the study looks outward. A selective review of the literature suggests that well-run pipelines streamline development, nudge teams to collaborate earlier, and keep quality from being an afterthought. That said, the barriers are real: skill gaps, tool sprawl, and a kind of cultural inertia where "the release" is still a big ceremony. CI/CD is not a silver bullet, but it is a lever.

The contribution here is practical. It documents how a complex system like DOMUS can adopt CI/CD in stages, how to measure progress with maturity criteria, and where teams typically get stuck. In short, the work offers a grounded roadmap for organizations that want DevOps not as a slogan but as daily practice.

Keywords: CI/CD, DevOps, DOMUS, Continuous Integration, Continuous Deployment, CI/CD Maturity Model

Resumo

Esta dissertação explora a implementação e avaliação de uma pipeline de CI/CD no contexto do DOMUS, uma solução de software de grande escala desenvolvida no P.PORTO. O DOMUS, composto por múltiplos componentes de software, enfrenta desafios devido à ausência de práticas de CI/CD, levando a ineficiências nos processos de versionamento, gestão de dependências e deployment. Os principais objectivos deste trabalho são conceber e implementar um processo de CI/CD adaptado aos requisitos do DOMUS e avaliar a sua eficácia utilizando o modelo de maturidade CI/CD.

Utilizando uma revisão exaustiva da literatura existente, a investigação salienta o potencial das condutas de CI/CD para melhorar os fluxos de trabalho de desenvolvimento, aumentar a colaboração e garantir a qualidade do software, ao mesmo tempo que aborda as barreiras de implementação, tais como lacunas de competências e resistência cultural.

Esta dissertação contribui para a compreensão das aplicações de CI/CD em sistemas complexos, oferecendo um "roadmap" para as organizações que pretendem melhorar os seus processos de desenvolvimento de software através de práticas DevOps.

Palavras-chave: CI/CD, DevOps, DOMUS, Continuous Integration, Continuous Deployment, CI/CD Maturity Model

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

CD3M	Continuous Delivery 3.0 Maturity Model.
CDMM	Continuous Delivery Maturity Model.
CI	Continuous Integration.
CI/CD	Continuous Integration and Continuous Delivery/Deployment.
DORA	DevOps Research and Assessment.
DSR	Design Science Research.
ISEP	Instituto Politécnico do Porto.
IT	Information Technology.
NISI	Netherlands National Institute for the Software Industry.
P.PORTO	Polytechnic Institute of Porto's.
PRISMA	Preferred Reporting Items for Systematic reviews and Meta-Analyses.
SHA	Secure Hash Algorithm.
UML	Unified Modeling Language.

Chapter 1

Introduction

The realm of software development has undergone a transformative shift, with automation and iterative methodologies becoming pivotal in addressing the complexities of modern applications. In this context, Continuous Integration and Continuous Delivery/Deployment (CI/CD) have emerged as essential practices for streamlining development workflows, enhancing collaboration, and ensuring software quality. This dissertation explores the implementation and evaluation of a CI/CD pipeline within the framework of DOMUS, Polytechnic Institute of Porto's (P.PORTO) official Information Technology (IT) platform.

Although its basic architecture has mainly remained, DOMUS was initially introduced in 2004 with a more limited functional scope and started out as an internal portal at Instituto Politécnico do Porto (ISEP). The platform was gradually moved to ASP.NET Web Forms (ASPX) from classic ASP (ASP 2.0). The majority of its modules are now ASPX-based, but some legacy ASP pages are still in use. VB .NET and C# are used for server-side development, while JavaScript with jQuery is used for client-side scripting, which was mainly introduced during the ASPX transition. Rendering and logic were almost exclusively handled on the server during the previous ASP phase.

1.1 Problem

DOMUS is the P.PORTO's official IT platform for supporting teaching and non-teaching activities [1]. Among the various functionalities, the following stand out:

- Viewing study plans, course unit sheets.
- Viewing timetables and summaries.
- Posting/viewing grades.
- Paying tuition fees.
- Exam registrations.
- Enrolment and enrolment renewal.
- Requesting certificates/diplomas.
- Requesting status.

- Applications to the various services.
- Justifying absences, requesting holidays, assignments.
- Attendance declarations for assessment tests.
- Booking meals.
- Scheduling face-to-face appointments.

This large-scale software solution comprises multiple software components, yet its development and maintenance processes face significant challenges due to the absence of CI/CD practices. The workload for developers is greatly increased when CI/CD is not used because versioning, dependency management, requirements validation, and static code analysis become manual and prone to errors. Furthermore, deployment—which is frequently the last phase of the software's lifecycle—takes longer than expected and is prone to human error.

The application of crucial regression testing procedures is also hampered by the lack of CI/CD. Regression testing makes sure that current features have not been negatively impacted by recent code changes. Regression tests must be carried out manually in the absence of automated pipelines, which raises the possibility of missed flaws and delays in problem identification. Furthermore, manual regression testing uses a lot of resources and frequently necessitates a large amount of developer time and coordination.

Since developers are overburdened with repetitive tasks, this lack of automation causes a bottleneck in the development lifecycle. Software reliability is also impacted by this absence, as there may be an increase in downtime or failures as a result of hidden problems.

These issues can be resolved by automating procedures like version control, dependency checks, static code analysis, and regression testing through the implementation of a CI/CD pipeline customized to the needs of the system. While continuous testing as part of the CI/CD workflow guarantees prompt defect detection and resolution, automation improves consistency and lowers human error. These improvements would streamline deployment processes and enhance overall software quality, reducing developer effort and improving efficiency.

1.2 Goals

This work's main objective is to create CI/CD for some software components of DOMUS to facilitate the job of the developers. This main objective can be divided into two parts:

- G1: Create a new CI/CD process for the DOMUS application with a solution designed using Unified Modeling Language (UML) Diagrams.
- G2: Evaluate the created CI/CD process using the CI/CD maturity model and DevOps Research and Assessment (DORA) Metrics.

In order to complete these goals, CI/CD and DevOps studies will be selected. Firstly, the necessary tools to develop the CI/CD process will be studied. After that, tools that exist in the current landscape will be compared, to see which ones fit the already existing DOMUS software solution and its dependencies. Finally, the CI/CD process will be created and evaluated.

1.3 Research Questions

Considering the goals of this dissertation, the Research Question was created:

- How can DevOps contribute to mitigate the inefficiencies of current DOMUS lifecycle?

1.4 Research Methodology

In this dissertation, Design Science Research (DSR) will be the research methodology used.

DSR is a methodological approach widely used in information systems and related fields to create and evaluate artifacts designed to solve identified problems. The core aim of DSR is to advance both the knowledge of the phenomena and the understanding of the solutions proposed. It is characterized by its focus on designing and building artifacts, such as models, methods, or software systems, and assessing them through rigorous evaluation [2].

As shown in Figure 1.1, DSR typically follows a cyclical process that includes the following stages:

- Problem Identification and Motivation: This initial phase involves identifying a problem or an opportunity for improvement in practice. The motivation for solving this problem is substantiated by demonstrating the significance and impact of the issue. This stage is located in the Chapter 1.
- Objective of the Solution: Based on the problem, researchers define the specific objectives that the solution artifact should achieve. These objectives guide the design and construction of the artifact. This stage is located in the Chapter 4.
- Design and Development: In this phase, the artifact is designed and developed. This may involve designing a model, a method, a tool, or any other form of artifact that addresses the problem. The design process is often iterative, allowing for refinement based on feedback and evaluation. This stage is located in the Chapter 5 and 6.
- Demonstration: The artifact is tested and demonstrated in a real-world or simulated environment to show its potential in addressing the problem. This could involve case studies, prototypes, or simulations. This stage is located in the Chapter 6.
- Evaluation: The artifact's effectiveness is evaluated using various metrics. This may include performance assessments, user feedback, or other forms of testing

to determine whether the artifact meets the objectives outlined in the previous stage. This stage is located in the Chapter 6.

- Communication: The academic community and practitioners are informed about the outcomes of the design process, including the artifact and its assessment. This stage makes sure that information is shared, which may result in the artifact being adopted and improved even more. This stage is located in the Chapter 7.

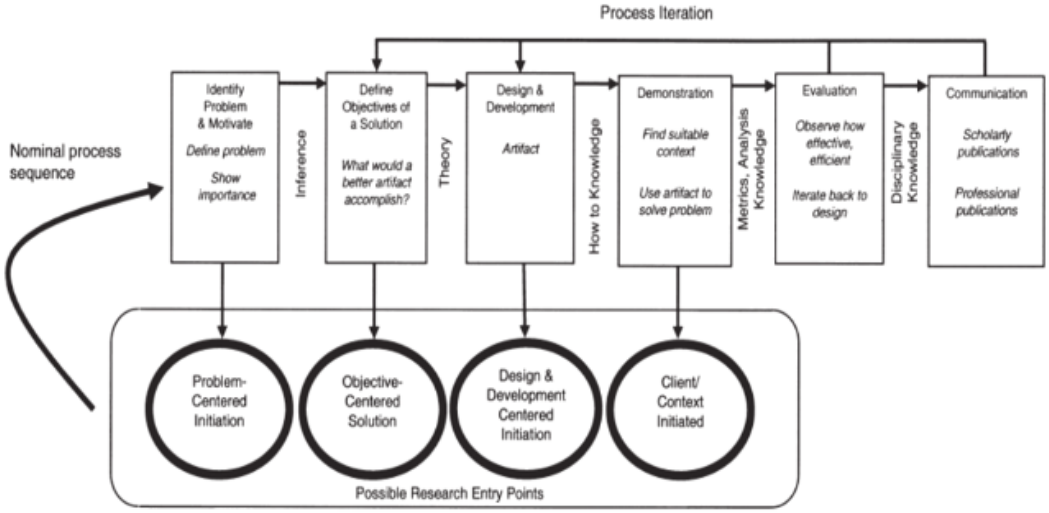

FIGURE 1.1: DSR Methodology Process Model

DSR is inherently iterative, meaning that it often involves cycles of refinement and testing, with each cycle bringing the artifact closer to optimal performance. Through this process, DSR not only contributes to solving practical problems but also generates new knowledge that advances the theoretical understanding of the domain. The methodology emphasizes the creation of artifacts that are both useful in practice and grounded in rigorous scientific inquiry.

1.5 Document Structure

The dissertation begins with the **Introduction**, which provides an overview of the problem, the goals of the research, and the research questions. This chapter also describes the research methodology and outlines the structure of the document, setting the foundation for the study.

The next chapter, **Skills Management and Project Planning**, evaluates the skills necessary for the successful implementation of the CI/CD pipeline and the project's overall objectives. It includes a skills gap analysis, strategies for development, and their impact on project outcomes. Additionally, this chapter outlines project planning processes, such as stakeholder analysis, risk assessment, and cost considerations, ensuring a structured approach to achieving the dissertation goals.

The third chapter, **State of the Art**, explores the foundational concepts and methodologies related to the dissertation. It discusses the context of the study,

background information on DevOps and CI/CD, and details on DORA metrics and available CI/CD platforms. This chapter also includes the research methodology, keywords, inclusion and exclusion criteria, and a systematic review process exploring the created Research Question.

Analysis, the fourth chapter, looks at the project from both a technical and business standpoint. It talks about the system's functional and non-functional requirements and assesses the business value of integrating CI/CD into DOMUS. A summary of the main conclusions wraps up the chapter.

The design strategy used for the CI/CD pipeline implementation for the DOMUS platform is presented in the fifth chapter, **Design**. It describes the deployment mechanisms, multibranch pipeline model, and architectural choices made to satisfy the functional and non-functional requirements of the project.

The technological stack used to implement the CI/CD pipeline, the orchestration process, and the constraints and limitations encountered are all covered in the sixth chapter, **Experimentation**. The Jenkinsfile setup, build, test, and static code analysis procedures, as well as the multibranch pipeline configuration and implementation outcomes, are also presented.

The seventh chapter, **Conclusion**, provides a summary of the research findings, answers the research questions, talks about the dissertation's contributions, and considers its shortcomings and potential future directions.

Chapter 2

Skills Management and Project Planning

This chapter focuses on two fundamental aspects essential for the successful execution of any academic or professional endeavor: the evaluation and development of skills required for the project and the strategic organization of resources and tasks through effective planning.

This chapter emphasizes how crucial it is to pinpoint skill gaps, develop focused improvement plans, and comprehend how these affect the project's outcome. It also examines the need for careful project planning, taking into account important elements like risk management, cost analysis, and stakeholder involvement.

By integrating these elements, the chapter aims to establish a solid foundation for the subsequent phases of the dissertation, ensuring that all components align with the overarching objectives and timelines.

2.1 Skills Management

This section evaluates the student's existing competencies, highlighting both the skills already acquired and those requiring further development. This analysis forms the basis for crafting an action plan to address the skills essential for this project. The skills matrix, located in the Table 2.1, outlines the skill name, the required level of proficiency, the current level of proficiency, the identified gaps, and comments regarding those gaps.

The following columns in the matrix provide critical insights into various aspects of skill assessment:

- **Required Proficiency**: This reflects the proficiency level on a scale from 1 (low proficiency needed) to 5 (high proficiency needed).
- **Current Proficiency**: This reflects the current proficiency level and is rated on a scale from 1 to 5.
- **Gap**: This gauges the difference between the Required Proficiency and the Current Proficiency, it can have 3 levels: None, Low, and High.

Table 2.1: Skills Matrix

Skill	Required Proficiency	Current Proficiency	Gap	Comments
CI/CD Pipeline Tools	4	3	Low	Good theoretical knowledge. Academic and professional experience with some of the available Pipeline tools.
DORA Metrics Analysis	5	2	High	Familiar with theoretical aspects; requires practical experience in measuring and interpreting metrics for performance improvements.
DevOps Practices	4	3	Low	Academic and professional experience in DevOps processes.
Version Control (Git)	4	4	None	Solid understanding and experience with version control systems through academic and professional projects.
Documentation Writing	4	3	Low	Ability to write clear documentation.
Static Code Analysis Tools	4	2	High	Basic familiarity with tools, with academic experience in configuring them. Needs more in-depth application and collaboration in real-world environments.
Automated Testing Frameworks	4	3	Low	Exposure to automated testing in academic and professional environments.
Project Management	4	3	Low	Limited to personal project management and Academic group projects; needs improvement in team coordination and resource management.
Written Communication	4	2	High	Average ability to document technical processes and produce academic content; improvement needed in clarity and conciseness.
Oral Communication	4	3	Low	Proficient in presenting technical topics; minor improvement needed in tailoring content for non-technical audiences and overall improvements.

2.1.1 Identification of Required Skills

The successful execution of this dissertation on CI/CD pipeline implementation and evaluation requires a diverse set of technical, analytical, communication, and management skills. Proficiency in CI/CD pipeline tools is essential for designing and automating the deployment processes, while familiarity with DevOps practices is critical for integrating development and operations workflows effectively. Additionally, an understanding of DORA metrics is necessary for evaluating pipeline performance and driving process improvements, although practical experience in applying these metrics is an area requiring further development.

Expertise in version control systems like Git plays a pivotal role in ensuring efficient collaboration and versioning during the project. Documentation writing skills are essential for creating clear and comprehensive technical reports that meet the

academic standards of the dissertation. Proficiency in static code analysis tools and automated testing frameworks is essential for validating software reliability and quality, as well as ensuring compliance with established standards.

Effective project management skills are also essential for planning and overseeing the different stages of the dissertation, such as pipeline design, tool configuration, and performance assessment. Lastly, clear communication of technical findings and results, whether through presentations, documentation, or conversations with academic and professional audiences, requires strong written and verbal communication abilities. Filling in small gaps in these areas will improve the dissertation's overall caliber and impact.

2.1.2 Assessment of Current Skills

An assessment of current skills reveals a mix of strengths and areas for improvement, as outlined in the Skills Matrix. The student demonstrates strong proficiency in Version Control (Git), supported by both academic and professional experience, providing a reliable foundation for collaborative and iterative development processes. This expertise ensures seamless integration of version control practices within the CI/CD workflows essential to the project.

The student shows developing proficiency in CI/CD Pipeline Tools and DevOps Practices, with a solid understanding of theoretical principles and experience with tools. However, further hands-on practice is needed to enhance proficiency in configuring and managing complex pipelines and effectively aligning DevOps methodologies with the project's requirements.

There is still opportunity to gain more hands-on experience configuring complex testing scenarios within continuous integration and deployment workflows, even though expertise in Automated Testing Frameworks is well-developed and has been exposed to both academic and professional settings. On the other hand, knowledge of static code analysis tools is still restricted to rudimentary academic knowledge. To effectively use these tools to improve code quality and align with the project's larger goals, more involvement is required.

The assessment also highlights areas of significant improvement required in DORA Metrics Analysis. While theoretical knowledge is evident, practical application is necessary to accurately measure and interpret metrics. Addressing this gap will enable the student to evaluate pipeline performance comprehensively.

The student exhibits a moderate level of competence in oral communication and documentation writing, and it is evident that they can effectively communicate technical information. To improve clarity, conciseness, and adaptability for a range of audiences, refinement is necessary. Written communication has promise, but more work is needed to better document intricate technical procedures. Likewise, more practice in oral communication would be beneficial, especially when it comes to adapting technical presentations for audiences that are not experts.

Finally, Project Management skills are well-developed for smaller-scale academic and personal projects. Strengthening these skills will ensure efficient coordination and successful execution of the dissertation's various phases.

2.1.3 Strategies for Skill Development

To bridge the identified skill gaps, a targeted development plan has been formulated based on the current skills matrix.

For technical skills, hands-on practice with CI/CD Pipeline Tools will be prioritized. This will involve configuring and managing real-world pipeline scenarios in small-scale projects, which will also help to train DevOps Practices. Tutorials, official documentation, and community forums will serve as supplementary resources. Practical application during the dissertation's implementation phase will enhance familiarity and confidence with these tools.

To address gaps in DORA Metrics Analysis, focused efforts will be made to integrate these metrics into the evaluation of the CI/CD pipeline. Tutorials, case studies, and tools such as performance tracking dashboards will aid in understanding deployment frequency, lead time, and change failure rates. Regular application of these metrics will reinforce their practical use in assessing pipeline efficiency.

Through practical configuration and application in codebases pertinent to the DOMUS project, static code analysis tools will be thoroughly examined. Gaining a better understanding of tools' contribution to code quality can be achieved through experimentation. This process will be guided by resources like tool documentation and online training courses.

Technical documents, thesis chapters, and reports will be revised in order to improve written communication skills. Peer and dissertation supervisor feedback will be sought, with an emphasis on logical structure, clarity, and conciseness.

Practice sessions for oral communication will be held in order to present technical subjects to non-specialist audiences, such as the dissertation supervisor, friends, and family. Incorporating feedback will enhance confidence, clarity, and delivery. By guaranteeing thorough initial explanations, the intention is to decrease the quantity of follow-up questions during presentations. Monitoring these presentations' completeness and clarity over time will make it easier to assess their progress.

Lastly, by using tools like Jira or Trello to plan, monitor, and oversee the different stages of the dissertation, project management abilities will be cultivated. Meeting deadlines, managing dependencies, and allocating resources will all be prioritized. These abilities will be honed in a real-world setting with weekly progress reviews and bottleneck reflections.

2.1.4 Identification of Skills to Improve

The primary skills identified for improvement include DORA Metrics Analysis, Static Code Analysis Tools, and Written Communication. These skills are critical for successfully implementing and evaluating the CI/CD pipeline and presenting the findings effectively.

DORA Metrics Analysis is essential for evaluating the performance and efficiency of the implemented pipeline. While the student has a theoretical understanding of these metrics, practical experience in applying them to measure metrics related to

CI/CD performance is needed. Improving this skill will enable accurate analysis and meaningful insights into the effectiveness of the pipeline.

Static Code Analysis Tools are crucial for ensuring code quality and reliability in the CI/CD workflow. Although the student has basic familiarity with such tools, deeper engagement with their configuration and application is required to fully utilize their potential. Hands-on practice with tools will strengthen the student's ability to identify and address code quality issues.

On the communication front, Written Communication requires significant enhancement to produce clear, concise, and audience-tailored documentation. Improving this skill is vital for effectively presenting technical findings and ensuring that the dissertation is comprehensible to both technical and non-technical audiences.

2.1.5 Impact of Skills on Project Success

The successful development of the identified skills is directly linked to the project's outcomes, with each skill playing a vital role in achieving the dissertation objectives.

Proficiency in CI/CD Pipeline Tools is critical for designing and implementing the deployment pipelines required for the DOMUS system. These skills ensure smooth integration, testing, and deployment processes, reducing manual effort and increasing the efficiency of the development lifecycle. Enhancing this area will enable the successful automation of critical tasks and the delivery of robust pipelines tailored to project requirements.

Proficiency in DORA Metrics Analysis is necessary to assess how well the pipelines that have been put in place are working. These metrics offer information on lead time, change failure rates, deployment frequency, and service restoration time. By filling the skill gap in this area, it will be possible to analyze the pipeline's performance and conformity to industry standards more precisely, leading to improvements and recommendations that can be put into practice.

Maintaining high-quality code throughout the project requires an equal amount of familiarity with static code analysis tools. These tools offer insightful commentary on the complexity, maintainability, and adherence to best practices of code. By honing this ability, the student will make sure that problems with code quality are found and fixed, which will help ensure a stable and expandable implementation.

Clear and effective Written Communication plays a pivotal role in articulating technical findings in the dissertation. Improvements in this area will enhance the clarity and structure of the research document, ensuring it is comprehensible and impactful for academic and professional audiences. Refining this skill will increase the accessibility and value of the research.

In order to effectively present the dissertation findings during the presentation and discussions with stakeholders, oral communication is essential. By honing this ability, the student will be able to confidently communicate intricate concepts in a clear, succinct manner, reducing the need for additional clarification and proving a deep comprehension of the research.

Last but not least, a project's adherence to its planned timeline and deliverables depends on having strong project management abilities. This entails controlling risks, dependencies, and resource distribution during the various stages of the study. Improved project management skills will lay the groundwork for efficient coordination and a dissertation that is completed successfully.

2.2 Project Planning

This section outlines the stakeholders and describes the processes for planning and controlling the dissertation project. A thoroughly developed project plan plays a crucial role in enhancing the probability of success, emphasizing the significance of this section.

2.2.1 Stakeholders

A clear understanding of the stakeholders, their influence, and their level of interest is fundamental to aligning the research with their requirements and expectations.

The student and supervisor are key stakeholders, holding both significant influence and strong interest in the project. The student is responsible for leading the work, ensuring that its scope, methodology, and outcomes align with the intended research goals. The supervisor plays a pivotal role by offering essential guidance, ensuring the study maintains academic integrity, and contributes meaningfully to the field. The supervisor is also representing P.Porto since DOMUS is an application that belongs to P.Porto, which means that P.Porto is indirectly a stakeholder as well.

Software engineers and DevOps developers might show a reasonable level of interest in the study findings due to their potential to improve DevOps practices and to learn about CI/CD pipeline implementation. Nevertheless, their direct impact on the progress or direction of the dissertation remains relatively minimal.

Professionals in the field may offer insightful, useful viewpoints, especially when it comes to the adoption and application of CI/CD and DevOps techniques. Despite having little direct control over the research process, they have a stake in the results' usefulness. Similar to this, open-source communities can gain from knowledge that improves DevOps and CI/CD processes, encouraging teamwork and efficient automation. Their involvement can greatly contribute to the wider adoption and distribution of the research findings, despite their relatively small influence.

2.2.2 Costs

In the context of this dissertation, direct costs are minimal, as the research is conducted primarily within an academic framework. However, several indirect costs might be considered, particularly for tools and resources essential to the project's implementation and analysis. These include potential expenses for software licenses required for CI/CD pipeline tools, static code analysis tools, or deployment services. While many of these tools offer free versions or are accessible through institutional licenses, advanced features or full functionality may require additional costs.

Access to relevant academic literature or specialized resources not covered by the institution's repository may also incur expenses. Furthermore, if any cloud-based services or hosting solutions are utilized for testing and deploying the CI/CD pipeline, this could contribute to indirect costs.

Overall, the predicted costs are estimated to range between 50 and 150 euros, depending on the specific tools and resources required. This estimation considers the potential need for premium licenses or subscriptions to ensure the comprehensive evaluation and effective implementation of the project.

2.2.3 Assumptions and Restrictions

When undertaking this research on the implementation and evaluation of a CI/CD pipeline in the context of DOMUS, several assumptions and restrictions influence the design, execution, and scope of the study.

Assumptions

- **Suitability of DOMUS as a Case Study**: DOMUS, as the official IT platform for P.PORTO, is considered a suitable case study for evaluating the application of a CI/CD pipeline. Its complexity and the lack of existing CI/CD practices make it an ideal candidate to demonstrate the challenges and benefits of such an implementation.
- **Feasibility of Technological Integration**: It is assumed that the tools and platforms selected for the CI/CD pipeline will integrate seamlessly with the existing DOMUS infrastructure. This includes compatibility with software components, development environments, and deployment processes.
- **Adequacy of Time and Resources**: The research timeline and available resources are assumed to be sufficient to design, implement, and evaluate the CI/CD process effectively. This includes access to necessary software, hardware, and expertise for conducting the study.
- **Effectiveness of the Evaluation Framework**: The CI/CD maturity model and DORA metrics are assumed to provide a reliable and comprehensive evaluation framework for assessing the pipeline's success. These metrics are expected to align with the research objectives and provide actionable insights.
- **Stability of External Factors**: It is presumed that there will be no major disruptions from external factors, such as changes in institutional priorities or unexpected technical challenges, that could hinder the progress of the research.

Restrictions

- **Limited Scope of Implementation**: Due to time constraints, most likely only a single CI/CD approach is implemented and evaluated. This limits the study's ability to compare multiple methods or address all possible scenarios within the DOMUS ecosystem.

- **Fixed Project Deadlines**: The research must adhere to strict academic deadlines for submission. This structured timeline restricts the flexibility to extend certain phases, potentially limiting the depth of experimentation and analysis.
- **Dependence on Institutional Policies**: The research must comply with institutional guidelines and policies regarding data usage, methodology, and ethical considerations. These requirements may impose additional constraints on the research process.
- **Focus on Software Development**: The study focuses primarily on the development processes of DOMUS and does not extensively explore other operational or administrative aspects of the platform.

2.2.4 Risk Assessment

Risk assessment and management play a pivotal role in the effective execution of projects, particularly within academic research contexts like dissertations. These processes ensure that potential obstacles are identified, evaluated, and mitigated before they can impact the project. Effective risk management safeguards the alignment of objectives, timelines, and resource allocation, reducing uncertainty and enhancing the reliability of project outcomes. By proactively addressing risks, researchers maintain control over their work environment, fostering a structured pathway to achieving high-quality deliverables and meeting project milestones [3].

A core instrument in managing risks is the Risk Matrix. This matrix evaluates the gravity of potential risks using two primary dimensions:

- **Probability**: This reflects the chances of a particular risk materializing, rated on a scale from 1 (very unlikely) to 5 (very likely).
- **Impact**: This gauges the possible effects of a risk should it occur, with scores ranging from 1 (negligible impact) to 5 (severe impact).

The combination of these two dimensions results in a PI Score, which classifies risks into four categories, as represented in the Figure 2.1:

- **Low Risk (Green)**: Low probability or impact, requiring little to no intervention.
- **Medium Risk (Yellow)**: Needs regular monitoring and some mitigation measures.
- **Medium-High Risk (Orange)**: Calls for active management and strategies to lower its likelihood or impact.
- **High Risk (Red)**: Represents critical risks needing urgent action and comprehensive response plans.

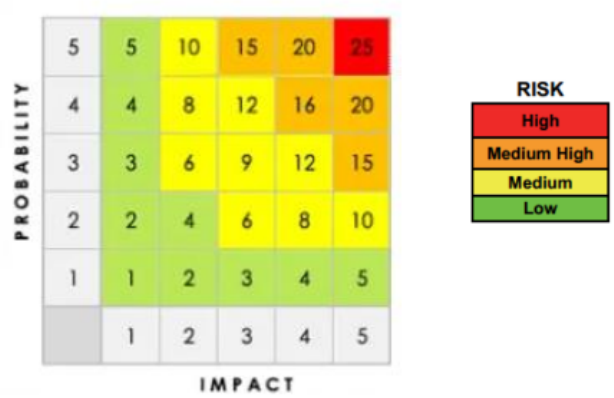

FIGURE 2.1: Risk Assessment Matrix

Table A.1 defined with risk identification and their result actions is available in Appendix A.

The risk table offers a detailed summary of potential challenges related to this dissertation project. It specifies each risk's nature, underlying causes, and potential impacts on the project's timeline and outcomes. Furthermore, it identifies the responsible party, assigns a Probability-Impact (PI) Score to rank the risks, and defines targeted response strategies.

Risks are categorized by severity, determined by their probability and impact, ensuring that critical issues receive prompt attention. Each response plan is customized for the respective risk, outlining measures like mitigation, avoidance, or contingency actions.

Considering the table A.1 we can identify two Medium-High risks that can cause trouble (ID 3 and 5).

Both ID 3 and 5 of table A.1 show how important it is to avoid issues regarding the DOMUS infrastructure compatibility with the necessary tools, which might cause problems with the implementation of CI/CD.

Chapter 3

State Of The Art

This chapter delves into the current state of the art, providing the necessary background and methodology to support the dissertation's goals. This chapter will be divided into four main sections.

The **Background** section delves into the theoretical underpinnings of DevOps and CI/CD, explaining their principles, practices, and significance in software development. It includes a detailed discussion of continuous integration, delivery, and deployment, highlighting their distinct roles in improving workflows and product quality, while also providing an overview of the most widely used CI/CD platforms. It discusses their features, benefits, and use cases, offering insights into their practical applications and relevance to the study.

The **Methodology** section explains the systematic approach used to gather and analyze data, including the use of peer-reviewed databases and the PRISMA framework. It outlines the procedures followed to guarantee a thorough and targeted review of the literature, the search query and key terms used to find pertinent studies, and the standards for choosing and rejecting studies. By identifying the articles used to address the research questions and offering a clear framework for the systematic review and the findings from the reviewed studies, it guarantees that only the most trustworthy and pertinent publications are included.

The **How can DevOps contribute to mitigate the inefficiencies of current DOMUS lifecycle?** section addresses the research question by analyzing the benefits and challenges of implementing DevOps. It explores how these practices impact software development and identifies common obstacles to adoption.

The **Summary** section provides a concise recap of the chapter, emphasizing its significance and linking it to the next chapter, which focuses on analyzing the current state of an existing application.

By laying a theoretical foundation and presenting the methodology for research, this chapter establishes the framework for examining the application of these principles and tools in subsequent sections of the dissertation.

3.1 Background

Considering the problem and goals of this dissertation, different topics will be analyzed for the Background.

3.1.1 DevOps

DevOps is a development methodology aimed at bridging the gap between Development and Operations, emphasizing communication and collaboration, continuous integration, quality assurance, and delivery with automated deployment utilizing a set of development practices [4].

DevOps practices significantly influence developers throughout the software development lifecycle in various ways:

- System Provenance Verification: Developers must verify a system's provenance during initialization. This ensures that the system has passed through the necessary approval gates and meets the required standards.
- Continuous Deployment: With continuous deployment, developers can directly push code to production without the need for coordination with other development teams. This impacts design decisions and the overall architectural approach.
- Environment Transitions: As systems progress through different environments on the way to production, developers must handle and manage configuration parameters effectively. This is crucial for maintaining system integrity and functionality.
- Post-Deployment Monitoring and Rollbacks: After deployment, systems are continuously monitored, and changes might be rolled back if issues arise. This influences the system's architecture, the data exposed, and the method of exposing such data.

Together, these practices influence design approaches, system management, and development workflows, mirroring the continuous integration and deployment paradigm promoted by DevOps. Furthermore, DevOps procedures heavily depend on a variety of tools, such as those for orchestration, continuous integration, container management, monitoring, deployment, and testing. Increasingly, software engineers are the ones who maintain and configure such tools [5].

3.1.2 Continuous Integration/Continuous Delivery

CI/CD, which stands for Continuous Integration and Continuous Delivery/Deployment, aims to streamline and accelerate the software development lifecycle [6].

CI/CD creates a faster, more precise way of combining the work of different people into 1 cohesive product. In DevOps environments, CI/CD streamlines application coding, testing, and deployment processes as seen in Figure 3.1. This gives teams a single repository for storing work and automation tools to combine and test the code to help ensure it works consistently [7].

FIGURE 3.1: CI/CD flow.

3.1.3 Continuous Integration

Continuous Integration (CI) in CI/CD refers to an automated process that allows developers to frequently merge their code changes back into a shared branch, or "trunk", while triggering automated testing to ensure the reliability of these changes. In modern application development, where multiple developers work on different features simultaneously, CI helps prevent the challenges of merging all code at once on a designated "merge day", which can be time-consuming and error-prone due to conflicts between changes. CI addresses these issues by enabling more frequent merges, reducing conflicts, and ensuring that all code changes are automatically tested through unit and integration tests. This approach makes it easier to identify and fix bugs quickly, improving the overall quality of the application.

3.1.4 Continuous Delivery

Continuous Delivery automates the release of validated code to a repository after the build, unit, and integration tests are completed in the Continuous Integration (CI) process. For effective continuous delivery, CI must already be integrated into the development pipeline [6]. In this process, each stage - from code merging to production-ready builds - includes test automation and code release automation. This allows the operations team to quickly deploy the application to production. Continuous delivery ensures that developers' changes are automatically bug-tested and uploaded to a repository (like GitHub or a container registry), where they can be easily deployed to live environments. Its primary goal is to maintain a codebase that is always ready for deployment with minimal effort required to release new updates, addressing the issue of poor communication and visibility between development and business teams.

3.1.5 Continuous Deployment

Continuous Deployment is the final stage in a mature CI/CD pipeline, extending continuous delivery by automating the release of code changes from the repository directly to production, where they become available to customers. This process addresses the challenge of overloading operations teams with manual tasks that slow down app delivery. Continuous deployment allows a developer's changes to go live within minutes of writing, assuming they pass automated tests, which facilitates quicker incorporation of user feedback. It reduces risk by enabling smaller, incremental releases rather than large updates. However, because there is no manual approval step before production, continuous deployment relies on robust test

automation, requiring significant upfront investment to ensure reliable testing and smooth releases.

3.1.6 CI/CD Maturity Model

A Continuous Delivery Maturity Model (CDMM) is a framework for assessing an organization's maturity in implementing continuous delivery practices [8]. It is intended to serve as a roadmap for businesses looking to enhance their software development workflow and eventually accomplish continuous delivery.

A Continuous Delivery Maturity Model is utilized for several reasons:

- Provides a framework for assessment: Organizations can assess their present continuous delivery procedures and pinpoint areas for development using the structured methodology that CDMM offers. This can assist organizations in determining their current position and the necessary actions to reach higher levels of maturity.
- Facilitates continuous improvement: Organizations can use CDMM as a guide to help them implement continuous delivery. This enables organizations to concentrate on a single area at a time and make incremental improvements over time.
- Increases quality: By assisting organizations with automated testing and validation, CDMM contributes to the improvement of software quality.
- Improves efficiency: Organizations can increase the frequency of releases and decrease lead times in their software development process by putting continuous delivery practices into place.

Continuous Delivery 3.0 Maturity Model (CD3M)

The Netherlands National Institute for the Software Industry (NISI) developed the CD3M, a framework for evaluating how mature an organization is in implementing continuous delivery practices [9]. It was developed in response to current software development trends and best practices, including DevOps and cloud native.

The CD3M can be divided in 5 different CD maturity levels [8]:

- Foundation: At this stage, the company has begun implementing some basic automation and has a basic understanding of continuous delivery. Planning, creating a continuous integration setting, and automating the build process are the main priorities.
- Novice: At this stage, the company is automating more of the development process and has started the process of continuous delivery. Enhancing software quality and incorporating continuous delivery into the development process are the main goals.
- Intermediate: At this stage, the company is incorporating more sophisticated testing and deployment procedures and has developed its continuous delivery

practices. Enhancing the software delivery process's speed and dependability as well as leveraging data and analytics to inform choices are the main goals.

- Advanced: At this stage, the company has incorporated continuous delivery into the entire software development process and has a high degree of automation. The emphasis is on applying sophisticated analytics and testing methods to guarantee the timely and dependable delivery of software.
- Expert: The company has attained a high degree of proficiency in continuous delivery at this point, and it is utilizing the newest instruments and methods to keep refining the procedure. The emphasis is on leveraging the newest technologies to support the delivery process and on leveraging real-time data and analytics to promote continuous improvement.

The CD3M also defines five categories that organizations can work on to improve their CD maturity [9]:

- Continuous Intelligence: At this level, the emphasis is on using analytics and data to make well-informed decisions regarding software development. It has features like telemetry, analytics, and real-time monitoring to learn more about the behavior and performance of systems.
- Continuous Planning: The planning and administration of software development are the main topics of this level. It has features like sprint planning, backlog management, and agile planning.
- Continuous Integration: The integration of code modifications and their testing are the main objectives of this level. It has features like code review, automated testing, and automated builds.
- Continuous Testing: Software testing is the main focus of this level. It has features like continuous testing, test-driven development, and automated testing.
- Continuous Deployment: This level is primarily concerned with software deployment to production. Among its features are automated deployment, blue-green deployment, and canary deployment.

Figure 3.2 shows how the five maturity levels and categories are used in order to evaluate a CI/CD implementation maturity.

	FOUNDATION Developing on a CD 3.0 platform, but the cycle is poorly automated.	NOVICE Working with basic automation on a reactive level.	INTERMEDIATE Running average CD 3.0 technologies with proactive elements.	ADVANCED Working with advanced CD 3.0 tech, that is quantitatively managed.	EXPERT Increasingly utilizing AI to improve the CD 3.0 development cycle.
INTELLIGENCE Making business decisions based on gathered used data.	Tracking customer behaviour on a server and receive feedback.	Basic monitoring of app usage and handling customer feedback	Advanced customer monitoring and A/B testing in place.	Receiving predefined metrics and reports. Decisions being made based on detailed analytics.	Real-time data collection analysis and reporting using AI
PLANNING Automating backlog item creation and prioritization to improve collaboration.	Managing the complete backlog on a centralized server.	Managing all work by means of a digital backlog.	Automatically creating items for the backlog.	Automatically receiving backlog prioritization suggestions.	Automatically prioritizing and creating backlog items using AI.
INTEGRATION Automatically building your software to shorten the development cycle.	Running a centralized version control system. Running a centralized build server	Running a workflow orchestrator and receiving reports. Running builds while the company is asleep.	Triggering builds after the commit of a new feature.	Running integrations on a scalable microservice architecture. Staged Integrations - compiling the source code that was edited only	Automatically scaling continuous integration services.
TESTING Automatically testing newly developed features to avoid tedious work.	Running a centralized unit-test server. Manually starting unit tests.	Running unit tests in a Continuous Delivery pipeline. Manually starting your automated integration tests.	Triggering integration tests in your Continuous Delivery pipeline. Manually starting your automated acceptance-tests.	Triggering acceptance tests in your Continuous Delivery pipeline. Manually starting your automated security and performance tests.	Triggering end-2-end regression tests in your pipeline.
DEPLOYMENT Automatically deploying new builds to scalable environments.	Running a centralized deployment server.	Running basic deployment scripts. Automatically deploying to a test server after a successful build.	Automatically deploying to the production server using a pipeline.	Automatically deploying without any downtime.	Automatically deploying on endless scalable platforms.

FIGURE 3.2: Continuous Delivery 3.0 Maturity Model.

3.1.7 DORA Metrics

DevOps Research and Assessment (DORA) offers a standardized framework of metrics to assess the effectiveness and maturity of DevOps processes. These metrics shed light on key aspects such as the speed of response to changes, the average deployment time for code, the frequency of updates, and insights into system failures.

DORA began as a dedicated team within Google Cloud with the mission of evaluating DevOps performance through a standardized set of metrics. The primary aim was to enhance performance, foster collaboration, and accelerate development processes. These metrics are designed as a tool for continuous improvement, enabling DevOps teams to establish performance goals, track progress, and refine their workflows effectively [10].

DevOps plays a vital role in ensuring the seamless operation of business software and processes, allowing users to focus on their tasks without disruption. DORA metrics are instrumental for DevOps teams in several ways:

- Delivering accurate and practical response time estimates.
- Enhancing work planning and execution.

- Highlighting areas that require improvement.
- Building consensus for strategic technical and resource allocation.

DORA metrics for DevOps teams concentrate on Four Key performance indicators:

- Deployment frequency.
- Lead Time for Changes.
- Change Failure Rate.
- Time to Restore Services.

Deployment frequency

Deployment frequency measures how often an organization successfully delivers code to production. It is one of the simplest metrics to track, as it requires only a single data table. However, determining deployment frequency involves more nuance than simply calculating deployment volume. For example, while it is easy to display daily deployment counts or average weekly deployments, deployment frequency specifically captures the consistency of deployments over time, not just their total number [10].

In the Four Keys system, Deployment Frequency is categorized into a "Daily" bucket if an organization deploys successfully on at least three separate working days per week. In other words, to qualify for daily deployment, most workdays must include a successful release. Similarly, weekly, monthly, or less frequent deployment buckets are assigned based on how consistently deployments occur over those periods [11].

Lead Time for Changes

Lead Time for Changes measures how long it takes for a code commit to be successfully deployed to production. To calculate this metric, you need two key timestamps: when the commit was made and when it was deployed [10].

Tracking this requires maintaining a record of all the changes included in each deployment. This can be efficiently achieved using triggers with a Secure Hash Algorithm (SHA) identifier that links back to the associated commits. Once there is a list of changes stored in the deployment table, it can be matched with the corresponding entries in the commits table to retrieve the timestamps. From there, the median lead time can be calculated to provide an accurate representation of this metric [11].

Change Failure Rate

Change Failure Rate represents the proportion of deployments that result in failures in production. Calculating this metric requires two key data points: the total number of deployments and the number of those deployments that led to production issues [10].

To derive this, the Four Keys system uses the deployment table to count all deployments and then correlates them with incidents. Incidents can be recorded through various sources, such as bug labels, forms linked to spreadsheets, or issue-tracking systems. The essential requirement is that each incident includes the deployment

ID, allowing it to be matched with the corresponding deployment entry. This linkage enables the calculation of the failure rate by comparing incidents to the total deployment count [11].

Time to Restore Services

Time to Restore Services measures how quickly an organization can resolve a production failure and restore normal operations. To calculate this, two key timestamps are needed: when the incident was reported and when it was resolved. Additionally, it is important to track the deployment associated with resolving the incident [10].

This data can be sourced from any incident management system, which should log the creation and resolution times for each incident. By linking incidents to their corresponding deployments, the metric can be accurately determined, reflecting the organization's responsiveness in addressing production issues [11].

3.1.8 CI/CD Orchestration

In this specific topic, we will filter the available CI/CD platforms by the most used ones. To see which ones are the most used, we will use the *Developer Ecosystem Report 2023* located in Figure 3.3 created by the JetBrains company [12].

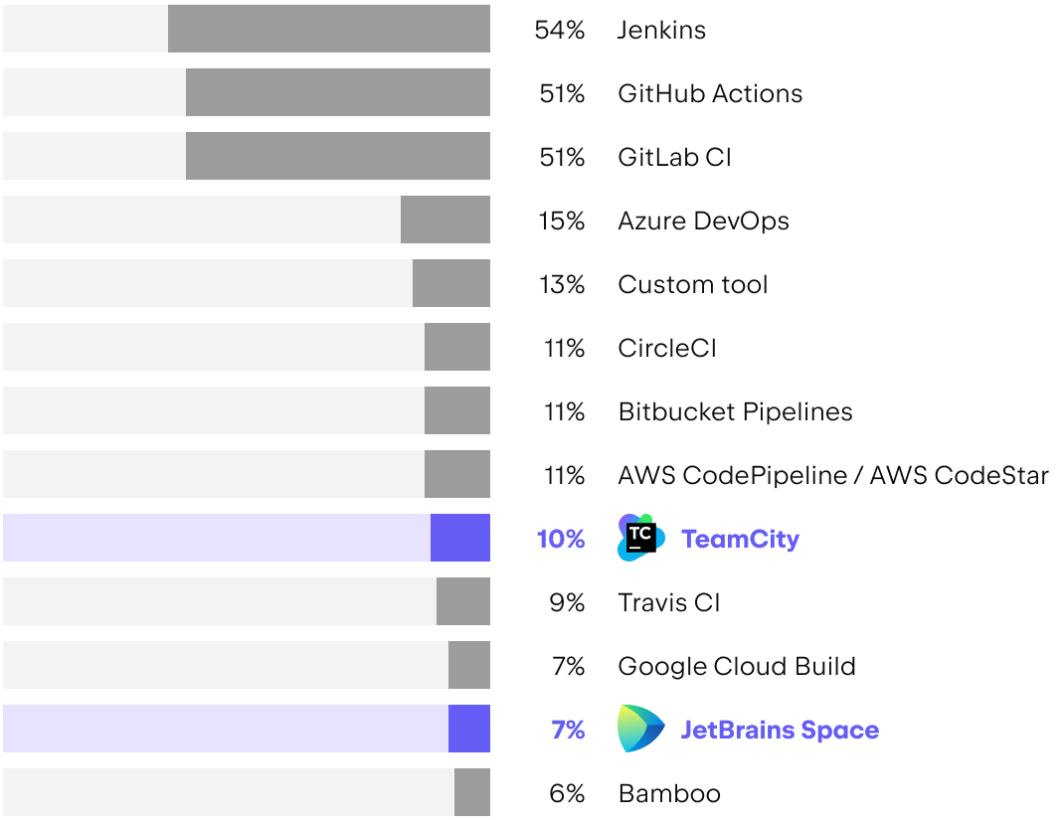

Figure 3.3: Most-used CI/CD platforms.

Jenkins

Jenkins is a self-contained, open-source automation server that can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software [13].

As a mature platform, Jenkins boasts an engaged and active community that supports its ongoing development and maintenance. It offers seamless integrations with all major version control systems and provides a vast selection of community-developed plugins, allowing for extensive customization of the Jenkins server [14].

GitHub Actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production [13].

GitHub Actions extends beyond traditional DevOps by enabling workflows to be triggered by various events in your repository. For instance, you can set up a workflow that automatically assigns the correct labels whenever a new issue is created.

GitHub offers virtual machines running Linux, Windows, and macOS to execute your workflows, but you also have the option to deploy self-hosted runners within your own data center or cloud environment [15].

GitLab CI/CD

GitLab CI/CD helps you realize the vision of software development that is iterative, tested, and always releasing. Measure developer productivity and watch it improve with one unified tool that encourages collaboration, immediate feedback, and bringing ideas to life—not manual, repetitive tasks [16].

GitLab CI seamlessly integrates with the broader GitLab platform, enabling automated pipelines for building, testing, and deploying code. While it is mainly designed to work with GitLab-hosted repositories, it can also be connected to GitHub, Bitbucket, and other Git servers. Besides using the provided hosted build runners, you have the option to run builds on self-hosted runners at no additional cost.

Azure DevOps

Azure Pipelines is the part of Azure DevOps that automatically builds, tests, and deploys code projects. Azure Pipelines combines continuous integration, continuous testing, and continuous delivery to build, test, and deliver your code to any destination. Azure Pipelines supports all major languages and project types [17].

Microsoft's Azure DevOps Pipelines is a component of the broader DevOps suite, it supports both cloud-hosted and on-premises build agents. Additionally, it offers built-in integrations for deployments across all major cloud service providers.

Bitbucket Pipelines

Bitbucket Pipelines is an integrated CI/CD service built into Bitbucket Cloud. It allows you to automatically build, test, and even deploy your code based on a configuration file in your repository [18].

Cloud-based containers are provisioned to meet specific requirements. Within these containers, commands can be executed on a local machine, while benefiting from a clean, customized environment that is configured to suit the user's needs. Bitbucket Pipelines integrates natively with the rest of the Atlassian suite, including Jira and Trello.

3.1.9 .NET

.NET is a secure, reliable, and high-performance application platform [19]. .NET is open-source, it allows to build apps with the .NET CLI, Visual Studio and Visual Studio Code, all while being cross-platform.

.NET Framework is the original implementation of .NET. It supports running websites, services, desktop apps, and more on Windows [20]. This .NET implementation is not cross-platform (Windows exclusive) and is not open-source. This version is not recommended to be used unless:

- The application building requires .NET Framework.
- The application that is being used requires .NET Framework.

3.1.10 Docker

Docker is an open platform for developing, shipping, and running applications [21]. Docker enables separation of applications from the infrastructure to enable quick software delivery.

The whole idea of Docker is to provide the ability to package and run application in a different environment called a container. This containers are lightweight and contain all necessary dependencies to run the application, so it is completely independent of software installed on the host machine.

Sharing containers is also done easily, so it allows fast deployment, since the container becomes the unit for distributing and testing the application.

3.1.11 SonarQube

SonarQube is an industry-standard on-premises automated code review and static analysis tool designed to detect coding issues in 30+ languages, frameworks and IaC platforms [22].

SonarQube directly integrated with the CI pipeline to check a project code against an extensive set of rules that cover many code attributes (maintainability, reliability, and security issues) on each merge/pull request.

Sonar products were designed in order to achieve the state of Clean Code. Clean Code is a standard for all code with the objective of creating secure, reliable and

maintainable software, so writing clean code is essential to maintain a healthy codebase.

3.2 Methodology

The searched content is based on conference papers and journal articles published in authentic electronic databases that are technically and scientifically reviewed by peers. The Data Sources used are the following:

- **DS1** - ACM Digital Library: `https://dl.acm.org/`
- **DS2** - IEEE Explore: `https://ieeexplore.ieee.org/Xplore/home.jsp`

The methodology that is being followed is the Preferred Reporting Items for Systematic reviews and Meta-Analyses (PRISMA).

3.2.1 Keywords and Search Query

In the following section, a search query will be created to find the most relevant studies. The main keywords used were Continuous Integration, Continuous Deployment, Continuous Delivery, CI/CD pipelines, DevOps, benefits and challenges.

```
(
    ("Continuous Integration"
    AND ("Continuous Deployment" OR "Continuous Delivery") OR "CI/CD
    pipelines" OR "DevOps")
    AND ("benefits" OR "advantages")
    AND ("challenges" OR "barriers" OR "limitations")
)
```

Listing 3.1: Search Query Used

The Trial-and-Error Search method was used. This method begins with an initial search query and progressively refines it based on the results obtained [23]. This approach was applied and the final search query is presented in the Listing 3.1. By strategically combining the AND and OR operators, the searches conducted across the selected data sources ensure that only publications containing the relevant queried content are retrieved.

3.2.2 Inclusion and Exclusion Criteria

The decided inclusion criteria are the following:

- **IC1** - Studies on the benefits and challenges of implementing CI/CD pipelines/DevOps.

The exclusion criteria for the articles are as follows:

- **EC1** - Studies that are not in English.
- **EC2** - Studies before 2017.
- **EC3** - Studies without an abstract or the full text available.
- **EC4** - Studies that do not focus on the DevOps area.

3.2.3 PRISMA Method

The PRISMA statement, first published in 2009, was developed to address inadequacies in the reporting of systematic reviews. It included a 27-item checklist outlining key elements to report in systematic reviews, accompanied by a detailed "explanation and elaboration" document that provided additional guidance and examples for each item. Since its release, PRISMA has gained significant recognition and adoption. It has been co-published in multiple journals, endorsed by nearly 200 journals and systematic review organizations, and widely used across various academic disciplines. Observational studies have shown that the adoption of PRISMA is linked to improvements in the comprehensiveness of systematic review reporting [24, 25].

PRISMA works by following three steps:

- **Identification**: Collect all studies that might be relevant to the research questions, ensuring they align with the established eligibility criteria.
- **Screening**: Evaluate the studies from the identification phase to determine if they meet the inclusion criteria. Initially, articles are classified as "Relevant" or "Irrelevant." In the second phase, the remaining articles are thoroughly reviewed to confirm their relevance to the research questions, discarding those that do not qualify.
- **Included**: Finalize the selection of articles that will be used to address the research questions.

By following these steps and applying the search queries alongside the inclusion and exclusion criteria, it becomes possible to collect the necessary information to address the research questions effectively.

3.2.4 PRISMA Method Results

Figure 3.4 illustrates the results of applying the PRISMA method to the selection process for studies included in this review:

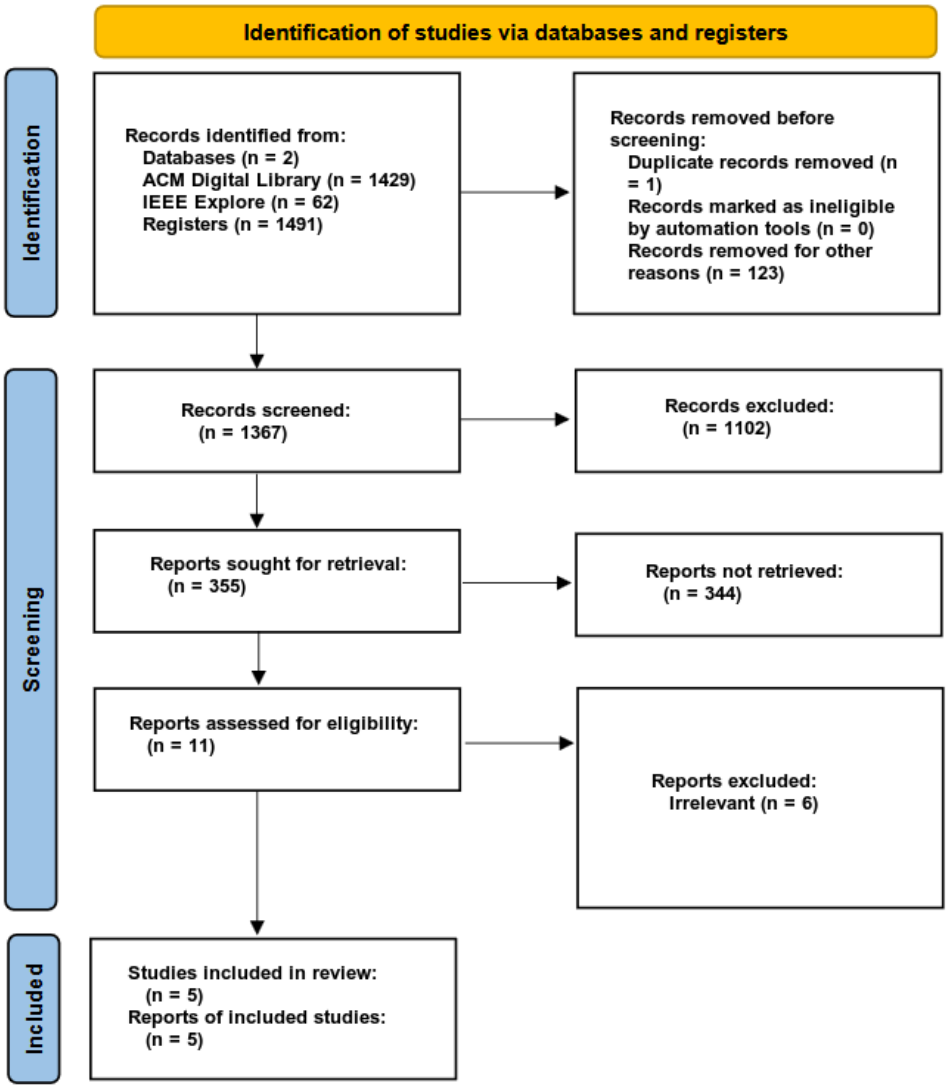

FIGURE 3.4: PRISMA method diagram

After completing the identification and screening process, the articles used to get information about the research question were [26–30].

3.3 How can DevOps contribute to mitigate the inefficiencies of current DOMUS lifecycle?

Continuous Integration (CI) and Continuous Deployment (CD) pipelines have become foundational practices in modern software development, enabling teams to streamline workflows, enhance collaboration, and deliver high-quality software efficiently. Before the adoption of DevOps, development and operations teams operated independently within organizations, each focusing on distinct objectives and goals. This separation often led to prolonged timelines for testing, design, and deployment

processes, ultimately resulting in customer dissatisfaction and increased stress for the teams involved [30].

3.3.1 Key benefits

DevOps builds upon the principles of agile by offering practical enhancements to existing agile practices. It bridges the gap between development and operations, ensuring that the iterative and collaborative nature of agile extends seamlessly into deployment and maintenance. By integrating continuous feedback and automation, one of the benefits of CI/CD is that it improves the efficiency and effectiveness of agile methodologies, driving smoother workflows and faster delivery cycles [27].

By implementing CI/CD you can also minimize manual repetitive work and be able to release whenever working software has been developed, which again leads to getting rapid user and customer feedback [26]. On top of that, this integration not only mitigates the time and manual effort traditionally required for identifying and rectifying software irregularities [29], and, as such, it also enhances code quality and reliability, while it also improves operation agility by minimizing delays and optimizing resource allocation, which are often challenges associated with traditional manual testing methods.

Most importantly, by encouraging cooperation between development and operations teams, utilizing their knowledge, encouraging automation, and establishing a collaborative culture, DevOps greatly enhances software delivery and product quality. Software development organizations looking to successfully deploy DevOps and establish a competitive and stable position in the rapidly evolving IT sector must acknowledge these benefits.

3.3.2 Challenges

Although DevOps has gained significant popularity, many software development organizations still struggle with its successful implementation due to an incomplete understanding of its processes and practices. Similarly, the concept of adopting DevOps and its implications remain ambiguous for some organizations [30].

The primary challenge for most software development organizations today is not deciding whether to adopt DevOps but determining how to implement it effectively and successfully. This process is complex and requires introducing changes to processes, personnel, and technology within the organization. Therefore, before embarking on the journey of DevOps adoption, organizations must thoroughly understand the techniques, strategies, and technologies that underpin this cultural transformation [30].

According to [30], the main challenges of DevOps adoption are the following:

- Lack of collaboration and communication.
- Lack of skills and experienced professionals.
- Confusion and lack of knowledge.
- Changing deep-seated company culture.

- Difficulty in achieving compatibility between DevOps and legacy systems.
- Replicate complex technology environments.
- Lack of top management support for DevOps Practices.
- Difficulties in implementing and using DevOps technology.
- Development and Operations teams work in isolation.
- Resistance to change.

The lack of collaboration and communication is an issue that often arises from the absence of clear communication frameworks, leading to difficulties in following instructions and coordinating among team members. To address this, operations teams must remain consistently integrated with other organizational processes while maintaining control. Additionally, IT operations and development teams must work closely and communicate effectively to achieve the continuous workflows that DevOps promotes [28].

Another major challenge frequently cited in the literature is the shortage of DevOps-related skills and the difficulty in finding experienced professionals. For instance, organizations often struggle with implementing testing processes, leading to subpar outcomes due to a lack of expertise. The transition to DevOps involves significant changes that extend beyond tools and procedures, requiring practical technological proficiency and specialized skills. Essential competencies include software development, infrastructure setup and deployment, post-deployment monitoring, problem-solving in infrastructure, and proficiency in using supporting tools [28].

DevOps demands expertise in both development (Dev) and operations (Ops), yet these areas require fundamentally different skill sets and knowledge bases, making it challenging for individuals to excel in both simultaneously. Consequently, having a workforce with the appropriate skills and expertise is vital, as the absence of skilled personnel can significantly hinder the successful adoption of DevOps [30].

Due to what was explained above, DevOps is a constantly evolving journey and not a final destination that can be reached quickly, which means that this journey will be unique to each software development organization.

3.4 Summary

This chapter provided an in-depth exploration of the current state of the art in DevOps and CI/CD methodologies, highlighting their significance in modern software development. The chapter began by outlining the structure and rationale for the sections, emphasizing their relevance to the dissertation's goals. The background section offered a comprehensive overview of DevOps principles, their impact on software development workflows, and the integral role of CI/CD pipelines in streamlining integration, testing, and deployment processes.

Detailed discussions on the distinctions between Continuous Integration, Continuous Delivery, and Continuous Deployment were presented, including their respective contributions to improving software quality and operational agility. The chapter also

examined what is the purpose of DORA metrics to analyze DevOps processes and examined the leading CI/CD platforms, such as Jenkins, GitHub Actions, GitLab CI/CD, Azure Pipelines, and Bitbucket Pipelines, providing insights into some of their functionalities, benefits, and usage scenarios.

This chapter introduced the systematic approach employed in this research, including the use of peer-reviewed databases and the PRISMA framework for identifying and analyzing relevant studies with the inclusion and exclusion criteria used to refine the research scope, ensuring a robust and focused literature review process.

In the next chapter, the focus shifts to analyzing the current state of an existing application. This transition ensures a logical progression by moving from the foundational knowledge and research methodology presented in this chapter to a detailed examination of how DevOps and CI/CD principles are applied in practice within a real-world context. This analysis will provide critical insights into the application's existing workflows, tools, and processes, laying the groundwork for identifying areas of improvement and aligning them with the goals of this dissertation.

Chapter 4

Analysis

In this chapter, we split the project into two general areas: the system requirements and the business value. We look at why the project has value, first to the business and then to users. We also develop a Business Model Canvas so that we understand clearly where the project sits in the big picture and what value it will provide. This will influence decisions that we will later make regarding technologies and methodologies.

Then we look at what the system is really required to do. We define the most important ideas in a domain model and enumerate the functional and non-functional requirements. By keeping the business objectives and the technicality apart, we keep the project in line and within limits and achieve actual value.

4.1 Business Value

The benefits of integrating CI/CD into DOMUS are presented in this section from two complementary perspectives. In order to achieve measurable efficiency and quality improvements at the platform level, it first looks at institutional outcomes, such as how automation can shorten release cycles, lower operational risk and rework, and match software delivery with P.PORTO's strategic and governance requirements. Second, it takes into account user outcomes, such as how improved dependability, quicker feature and fix turnaround, and more seamless releases result in a more responsive experience for administrators, academic staff, and students.

4.1.1 Business Perspective

DOMUS platform, the official IT platform of P.PORTO, plays a very central role in keeping a large multitude of academic and administrative processes operational, from maintaining study plans and tuition payments to exam enrollment. However, since DOMUS does not implement any Continuous Integration and Continuous Deployment (CI/CD) practices, it has bred enormous inefficiencies. Development work is very manual and results in versioning errors, delayed deployments, dependency management errors, and more opportunities for unknown bugs due to the lack of automated testing.

Imbibing a CI/CD pipeline ought to confer significant business benefit through the automating of these critical processes. This will minimize the occurrence of human mistakes, reduce deployment cycle time, boost the software reliability, and allow

developers to conduct value-additive work rather than mind-numbing maintenance activity. Additionally, with the ability to identify and fix defects with ease, the platform's quality and stability will rise, which an institution like a college or a university with thousands of students and teachers requires.

More broadly, adopting a CI/CD process that is aligned with DevOps methods will put DOMUS and P.PORTO at the forefront of forward-thinking, efficient, and robust in regards to their digital transformation strategy, allowing technological advancement to underpin the long-term academic and administrative goals of the institution.

4.1.2 User Perspective

From the user's point of view—students, academic staff, and administrative staff—improvement in the underlying development and deployment cycle will result in a more stable, responsive, and efficient system. Users will experience fewer service downtimes, faster access to new features and bug fixes, and overall increased platform stability.

Given that DOMUS is the foundation of critical scholarly and operational activities, increasing its reliability will have a direct implication on user satisfaction. Fast turnaround on bug fixes and improvements equate to being able to respond to issues raised by users promptly, thereby making the platform consistent with evolving user needs and expectations.

Moreover, by shortening the time and work developers need to dedicate to manual deployment and testing, DOMUS can also innovate more rapidly, releasing new services and enhancements that further enhance the user experience.

4.2 Requirements Engineering

In this section it will be explained what were the created functional and non-functional requirements.

4.2.1 Functional Requirements

There is no real actor, but let us assume the system as an actor for all functional requirements.

- **FR1**: The construction, testing, and deployment of DOMUS components will be automated using CI/CD pipelines.
- **FR2**: The system will incorporate the pipelines with a pre-existing version control system for auto-detection of changes in the code.
- **FR3**: The system will build the application in order to verify if there are any compilation issues.
- **FR4**: The system will perform automated regression testing so that new changes do not break current functionality.

- **FR5**: The system will carry out static code analysis as part of the CI process for maintaining code quality.
- **FR6**: The system will automatically deploy builds to production environments.

4.2.2 Non-functional Requirements

The non-functional requirements specify quality attributes the CI/CD pipeline and the supporting workflows have to fulfill. They ensure the technical soundness and usability of the solution by its users.

- **NFR1**: The system shall be compatible with the existing DOMUS infrastructure, requiring minimal changes to the underlying software components.
- **NFR2**: The solution should be scalable to accommodate future expansion to include other DOMUS components as necessary.
- **NFR3**: There will be documentation created for pipeline configuration, usage, and debugging to allow future maintenance and knowledge onboarding of new team members.
- **NFR4**: Gathering and storing of DORA metrics data will be in accordance with institutional data governance and privacy policies.
- **NFR5**: The pipeline framework used needs to be open-source and allow to be used on-premises.

4.3 Summary

In this chapter, the project was examined from both the business and technical perspectives. First, the business implications of introducing a CI/CD pipeline into the DOMUS platform were considered. From a business point of view, the project solves quite practical inefficiencies regarding manual versioning, testing, and deployment and provides automation as a path toward greater software quality, operational effectiveness, and reduced developer workload. From the point of view of users, the upgrades seek to provide a more responsive and stable platform, which will improve the experience of the students, teaching staff, as well as administrative personnel.

After the business analysis, this chapter specified the requirements engineering process. The domain model established the most significant notions applicable to the project, and the functional and non-functional requirements specified what must be provided by the CI/CD solution and the levels of quality it shall provide. The requirements form a basis for the subsequent design, implementation, and validation stages.

By setting out the business drivers and technical requirements, such analysis ensures the project remains focused on its strategic goals, and it is possible to devise a solution that is both effective and achievable.

Chapter 5

Design

In this chapter, we are going to present the design approach taken for the CI/CD pipeline implementation for the DOMUS platform. The design phase is a crucial connection between business requirements and needs analysis and solution development in practice.

We start by specifying the architecture that will underpin the automation of build, test, and deployment processes so that it addresses the project's functional and non-functional requirements. We make architectural choices using best practices in software architecture and DevOps and the application of standard notations and tools to improve clarity and maintainability.

Additionally, this chapter describes how the requirements elicited inform the architectural decisions so that the solution proposed is not only technologically feasible but also in accordance with the project goals. The decisions made here provide the basis for the subsequent implementation phase.

5.1 Architecture

The target design of the CI/CD pipeline of the DOMUS platform was planned to be robust, automated, and maintainable. It is a DevOps best practice-based approach, with modern tooling and patterns to enable seamless collaboration and high velocity of development. Taking into account **NFR5**, the selected pipeline automation server chosen was **Jenkins** because it was the only one that was open-source while also allowing on-premises use. The design is split into two primary components: the multibranch pipeline model, which manages code integration and testing, and the deployment mechanisms, which are separated as a concern point to provide flexibility and improved environment control. Each component is presented in detail in the sections that follow.

5.1.1 Multibranch pipeline

The pattern set out to create the CI/CD pipeline implementation of the DOMUS platform is a modular, extensible design based on a multibranch pipeline model.

Figure 5.1 shows the model that was selected in order to allow parallel feature development, to have strong testing between branches of code, and to automate deployment entirely.

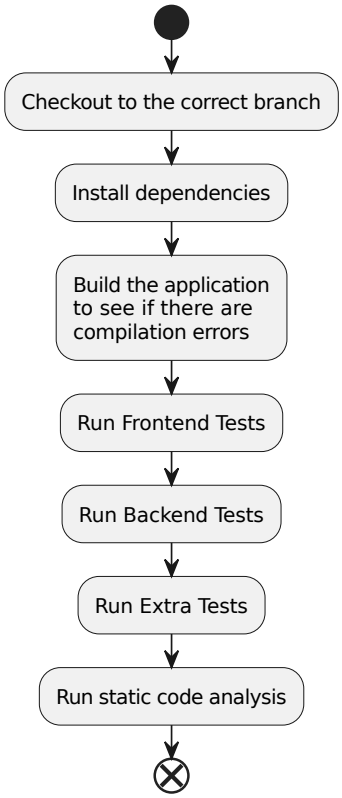

FIGURE 5.1: Multibranch pipeline

Pipeline runs are powered by Jenkins through its multibranch capability that automatically discovers new branches or pull requests in the repository and creates corresponding pipeline jobs. The same pipeline stages that are set up are utilized by all branches, with the same build, test, and deployment tasks in place across development streams.

The pipeline is a sequence of successive stages, often including:

- **Checkout**: Gets the most recent code from a specific branch.
- **Build**: Installs the necessary dependencies and compiles the application and packages.
- **Test**: Executes all necessary tests.
- **Static Analysis**: Performs code quality checks using SonarQube.

This layout supports feature branching and trunk-based development with space for team parallelism. The temporary branches in the pull request style with isolated build and tests enable early identification of integration issues. The desired characteristics promoted by this pattern are maintainability and traceability, and also promoting quick iteration cycles—main objectives for the DOMUS platform. It is an expression of DevOps best practices by combining automation, visibility, and policy management in one framework.

5.1.2 Deployment

The deployment stage is purposefully separated into a separate Jenkins job, apart from the multibranch CI pipeline, in order to meet the need that deployments be automated while still being secure, auditable, and compatible with DOMUS's diverse runtime environments. A fully automatic process is not really possible considering the release process present in DOMUS. So, the second best choice is to create a separate job that is triggered manually. While maintaining distinct operational boundaries between build/test operations and environment modifications, this division of responsibilities aligns the architecture with **FR6** (automatic deployment to target environments) and with **NFR1** (requiring minimal changes to the DOMUS infrastructure).

The deploy job is made to work with all DOMUS components. Per-environment variable files and parameterized configuration are used to handle environment differences, maintaining pipeline logic consistency while enabling each application to specify its own deployment recipe.

Listing 5.1 shows an example of a Jenkinsfile using parameterized configuration.

```
pipeline {
  agent any
  parameters {
    string(name: 'MODULE', description: 'DOMUS component')
    choice(name: 'ENV', choices: ['dev', 'qa', 'staging', 'prod'], description
    : 'Target environment')
    string(name: 'VERSION', defaultValue: 'latest', description: 'Image tag /
    version')
  }
  stages {
    stage('Deploy') {
      steps {
        echo "Component: ${params.MODULE}"
        echo "Environment: ${params.ENV}"
        echo "Version: ${params.VERSION}"

        // Deployment necessary commands using the passed parameters.
      }
    }
  }
}
```

Listing 5.1: Example of a deployment Jenkinsfile

By offering a clean, controlled path from a validated build to a running release, this method enhances the multibranch CI model previously discussed and maintains the deployment mechanism extensible and maintainable as more DOMUS projects adopt the pipeline.

5.2 Summary

The DOMUS CI/CD solution's architectural design is described in this chapter.

Jenkins is chosen as the core platform based on non-functional requirements, especially the requirement for an open-source, on-premises automation server. Two complementary elements form the framework of the architecture. Firstly, a multibranch continuous integration pipeline standardizes build, test, and static analysis

across all pull requests and branches, allowing for parallel feature development and enhancing visibility, maintainability, and early integration problem detection. Every change yields consistent, traceable results thanks to its standard stages, which include checkout, build, automated testing, and SonarQube-based static analysis.

Second, in order to meet security, auditability, and environment-specific requirements within DOMUS, deployment is specifically divided into its own Jenkins job. Deployments are initiated manually against verified artifacts due to the current release process's inability to promote them fully automatically. Per-environment configuration and parameterization guarantee cross-application reuse. As more DOMUS projects follow the pattern, this division of responsibilities maintains distinct operational boundaries between verification and release and offers a controlled, extensible path from a successful CI run to a running release.

Chapter 6

Experimentation

This chapter describes the technological stack that is used to implement your work.

This Implementation is done only on one of DOMUS repository. This will work like a Proof of Concept to implement on all other DOMUS repositories.

The base pipeline will be the same, since most of the DOMUS project are built using .NET Framework, what might change is to where the multibranch pipeline will point to and there is a possibility that small tweaks will need to be done in the pipeline.

6.1 Technological Stack

The tools used were the following:

- **.Net Framework**: In order to build and test the applications related to DOMUS.
- **Java**: Necessary to use Jenkins.
- **Jenkins**: Framework used to create the necessary pipeline configuration.
- **Git**: Tool necessary for version control of the application used for branching control on the pipeline.
- **Docker**: Container that will contain a Jenkins instance to run the pipeline.
- **SonarQube**: Framework necessary to implement static code analysis.

6.2 Orchestration

The implementation of requirements should have references to the functional and non-functional requirements described on section 4.2.

6.2.1 Restrictions and Limitations

The fact that most of DOMUS projects use .NET Framework 4, forces the machine that is running the pipeline to be using Windows, since .NET Framework is only compatible on Windows. This is important because it causes restrictions on the type

of OS we can use in order to comply with the non-functional requirements NFR1 and NFR2.

DOMUS has projects with dependencies with each other, while using Visual Studio they are able to get the dependencies when needed. This is not possible when running in the pipeline. So, in order to make the pipeline run, all necessary DLL files need to be included inside the repository that will be used by the pipeline.

6.2.2 Pipeline Creation

In an **ideal scenario**, Jenkins should be used using Docker to maintain it fully containerized. The steps to follow for that specific scenario are presented bellow.

To start we had to download Docker. After downloading it, Docker needs to be started and the Jenkins official Docker image had to be pulled. In order to use that image the command listed in the Listing 6.1 was used in the terminal:

```
1 docker pull jenkins/jenkins:lts-jdk17
```

Listing 6.1: Command used to pull the Jenkins image

After pulling the image, to start Jenkins the command listed in the Listing 6.2 was used:

```
1 docker run -v jenkins_home:/var/jenkins_home -p 8080:8080 -p
    50000:50000 --restart=on-failure jenkins/jenkins:lts-jdk17
```

Listing 6.2: Command used to start Jenkins

This command will start Jenkins in the 8080 port. The ports can be easily changed in the command by changing the value after the first flag.

Figure 6.1 shows a deployment diagram in order to explain the relationship between all machines/nodes and how they interact with each other.

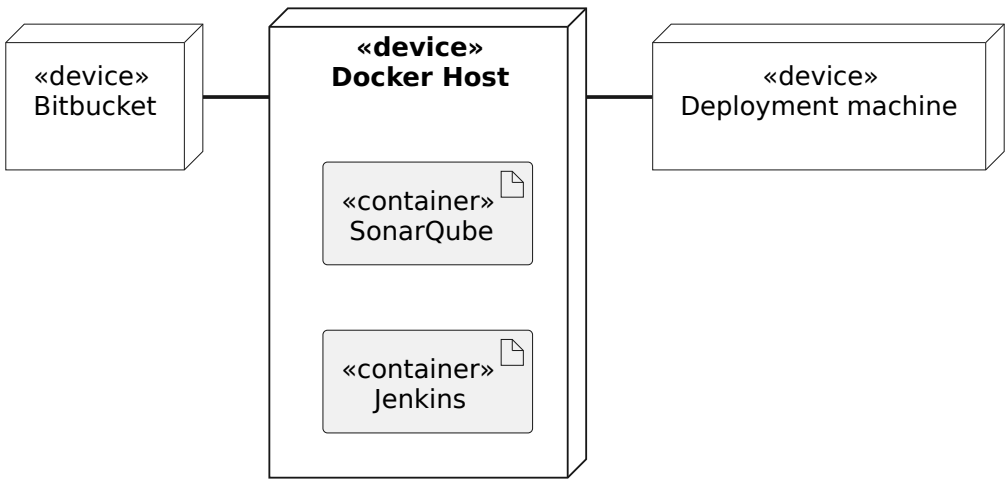

Figure 6.1: Deployment diagram

In **reality**, because of the limitation of having to use a local git repository, instead of one in the internet, Docker is not really an option since it does not allow to access local machine files unless we move those specific files inside the docker container. So, the Jenkins that was used was the WAR version one.

This WAR file is located in the official Jenkins download page. In order to run this WAR file, it is necessary to install Java JDK. The recommended version for Jenkins is JDK 17, so that was the one used.

To start the WAR file, the command specified in the Listing 6.3 was executed:

```
java "-Dhudson.plugins.git.GitSCM.ALLOW_LOCAL_CHECKOUT=true" -jar
    jenkins.war
```

Listing 6.3: Command used to start Jenkins

The flag **ALLOW LOCAL CHECKOUT** is necessary specifically for debug/local development to allow Jenkins to checkout to local branches, this part is not necessary when Jenkins is working on a production environment.

After running the last command, the default configuration shown in the Figure 6.2 was used.

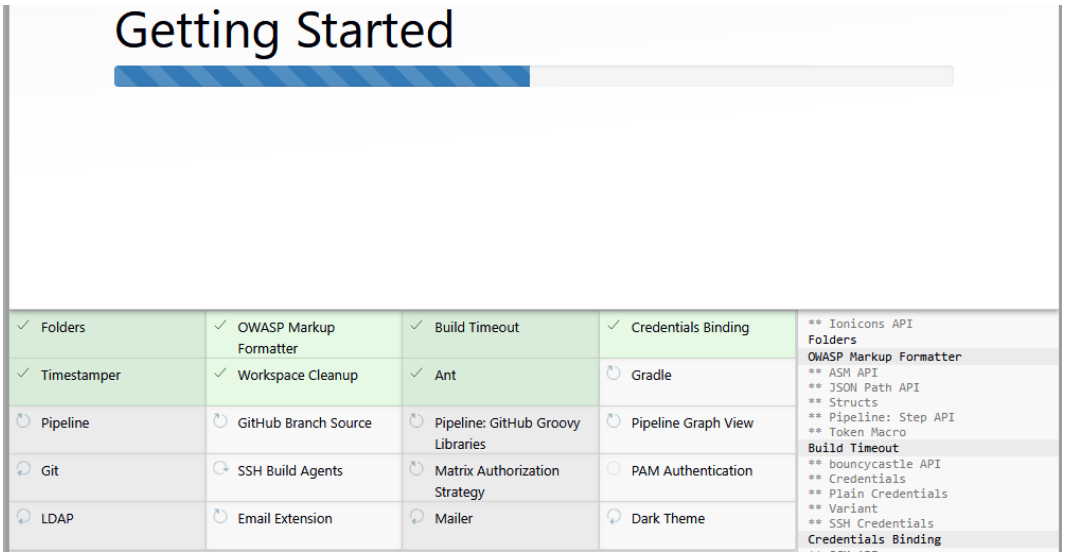

Figure 6.2: Jenkins default config

6.2.3 Configuration of Multibranch pipeline

This subsection will explain the steps needed to comply with the functional requirements FR1 and FR2.

After starting Jenkins, the pipeline needs to be created. To do this first we need to press the ***Create a job option*** as shown in the Figure 6.3.

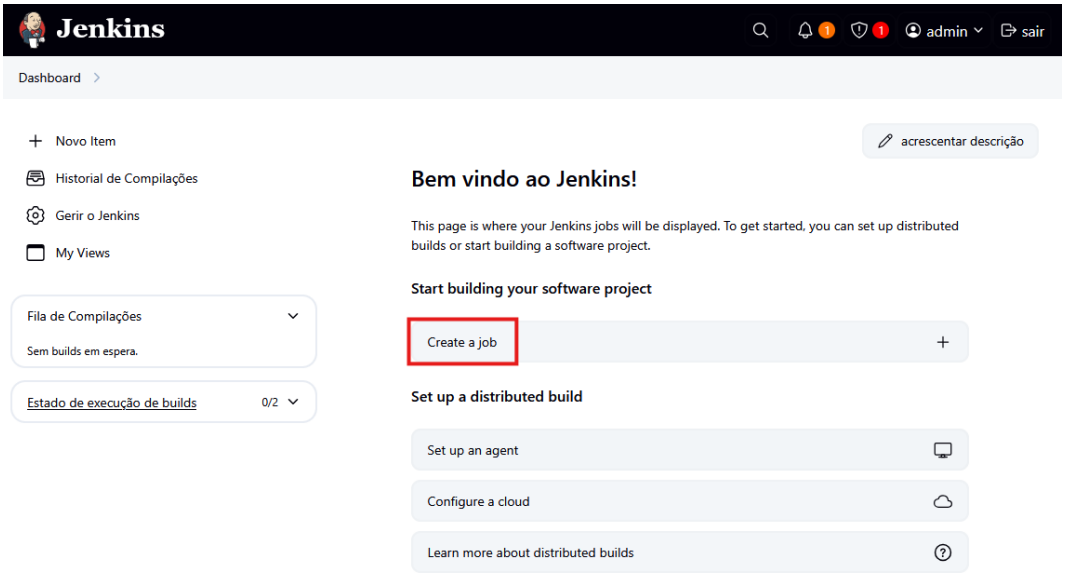

FIGURE 6.3: Jenkins Create a job

After that, select the option ***Pipeline*** as shown in the Figure 6.4

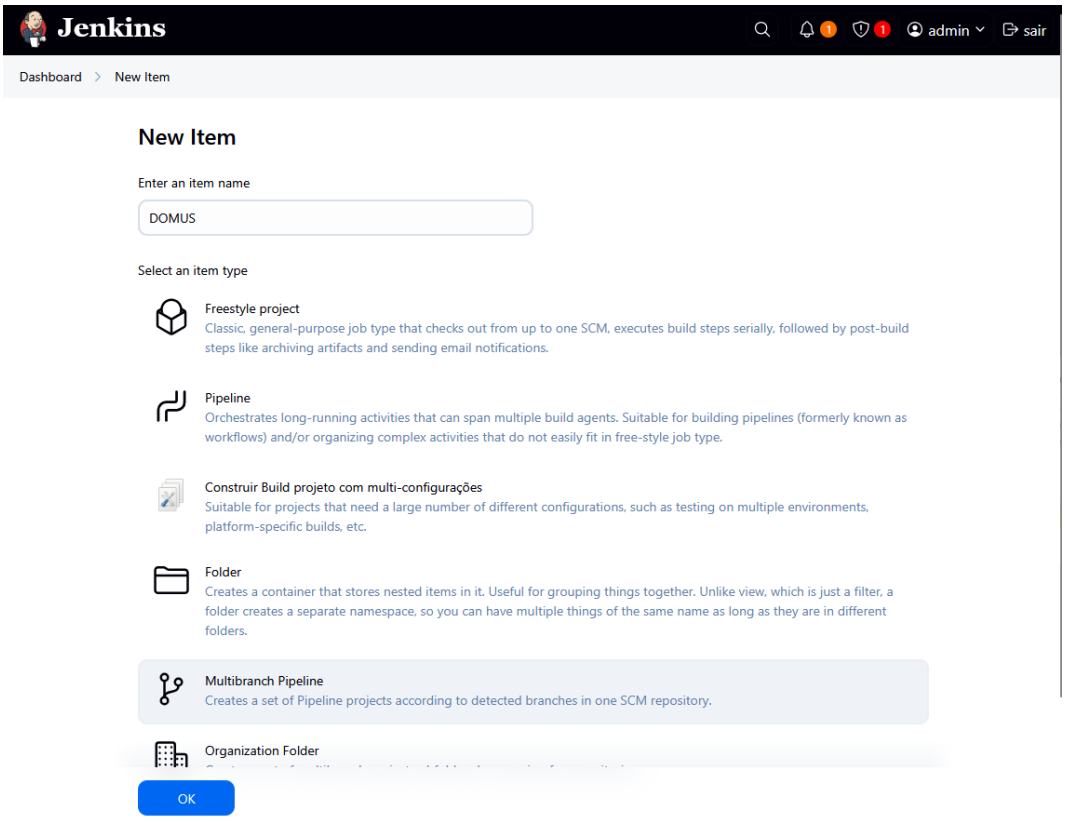

FIGURE 6.4: Jenkins Pipeline creation

After creating the Pipeline, we need to configure it. The only thing that really

matters to change here is to point to the correct git repository. The Figure 6.5 shows the configuration used for it:

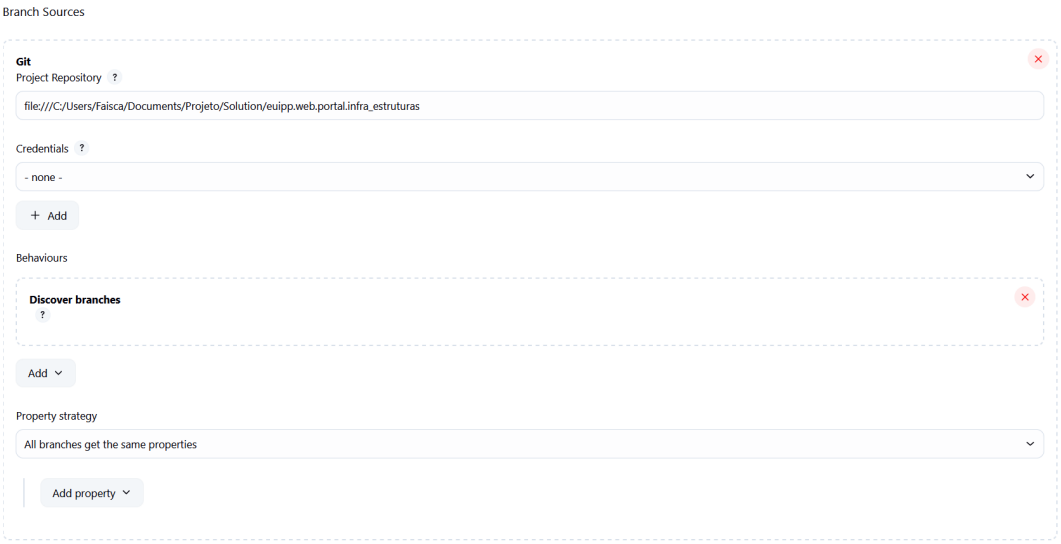

FIGURE 6.5: Jenkins Pipeline configuration

6.2.4 Jenkinsfile

In order to tell the pipeline the steps it needs to follow when it is executed, it is needed to create a Jenkinsfile. The Jenkinsfile is a text file that contains instructions on what the pipeline should do when a new execution is started. The location of this file can be specified in the configuration, but most of the time, it should be on the root of the project, where the .git folder is located, because in that specific scenario there is no need to do custom configuration.

In the Listing 6.4, there is an example of a Jenkinsfile.

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building...'
            }
        }
        stage('Test') {
            steps {
                echo 'Testing...'
            }
        }
        stage('Deploy') {
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

LISTING 6.4: Example of a Jenkinsfile

The next steps will explain how to configure the specific steps in the Jenkinsfile and the difficulties found in each step.

6.2.5 Build

This subsection will explain the steps needed to comply with the functional requirement FR3.

To build the application, it is not really possible to use something like an IDE, everything needs to be built using terminal or Jenkins plugins. Initially, .NET CLI was used to build the application.

The command used is the one show in the Listing 6.5.

```
dotnet build
```

Listing 6.5: Command used to start .NET CLI build

The result of the command used is shown in the Figure 6.6.

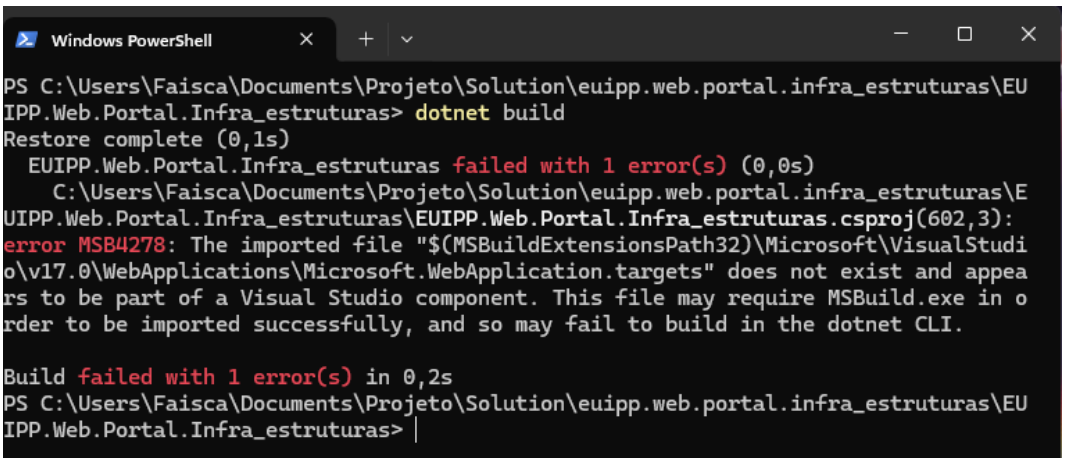

Figure 6.6: .NET CLI build error

As shown in Figure 6.6 this presented several errors. Later, after reading documentation, it was understood this is only supported for .NET Core, which made it impossible to use on DOMUS, since it uses .NET Framework.

After reading the official documentation of .NET it was noticed that in order to run the application, we where forced to use an exe util from Visual Studio. This exe is specifically created to run old .NET Framework applications. To run this exe, the command listed in the Listing 6.6 was used:

```
"C:\Program Files\Microsoft Visual Studio\2022\Community\MSBuild\
    Current\Bin\MSBuild.exe" /t:Rebuild"
```

Listing 6.6: Command used to start .NET CLI build

The results of the command are shown in the Figures 6.7 and 6.8:

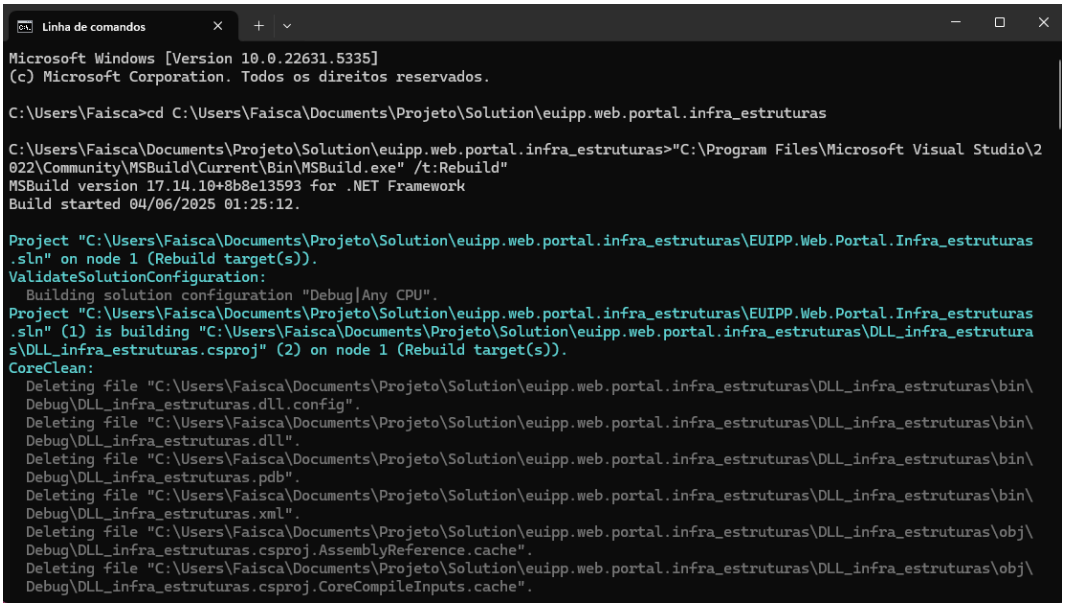

FIGURE 6.7: MSBuild part 1

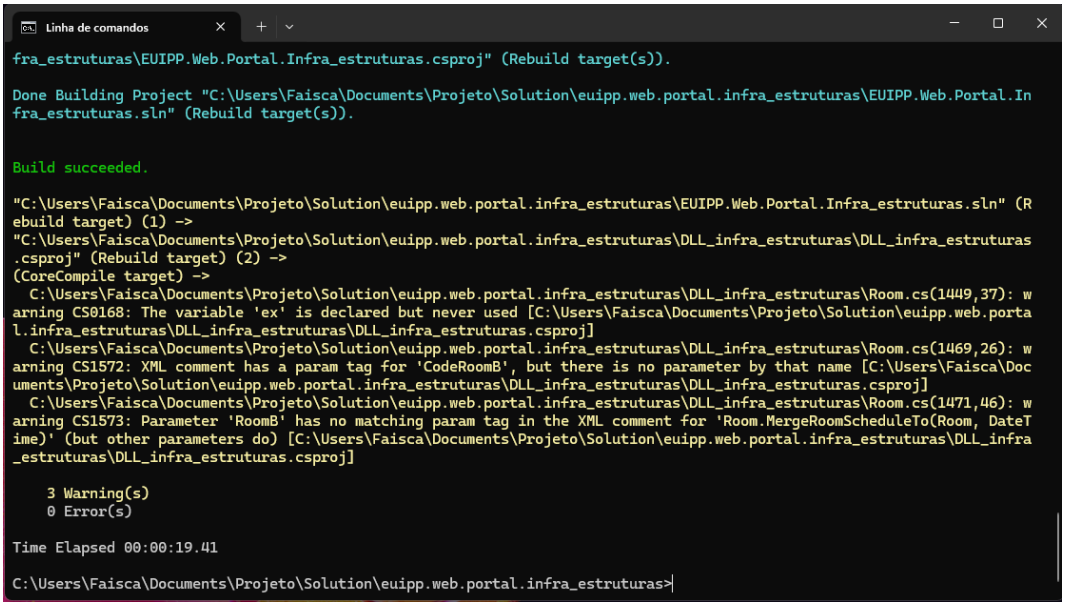

FIGURE 6.8: MSBuild part 2

Later, it was found that Jenkins actually had a fix for this specific problem. There is a plugin in Jenkins called **MSBuild** that allows us to use this util exe without actually having to download Visual Studio as shown in the Figure 6.9.

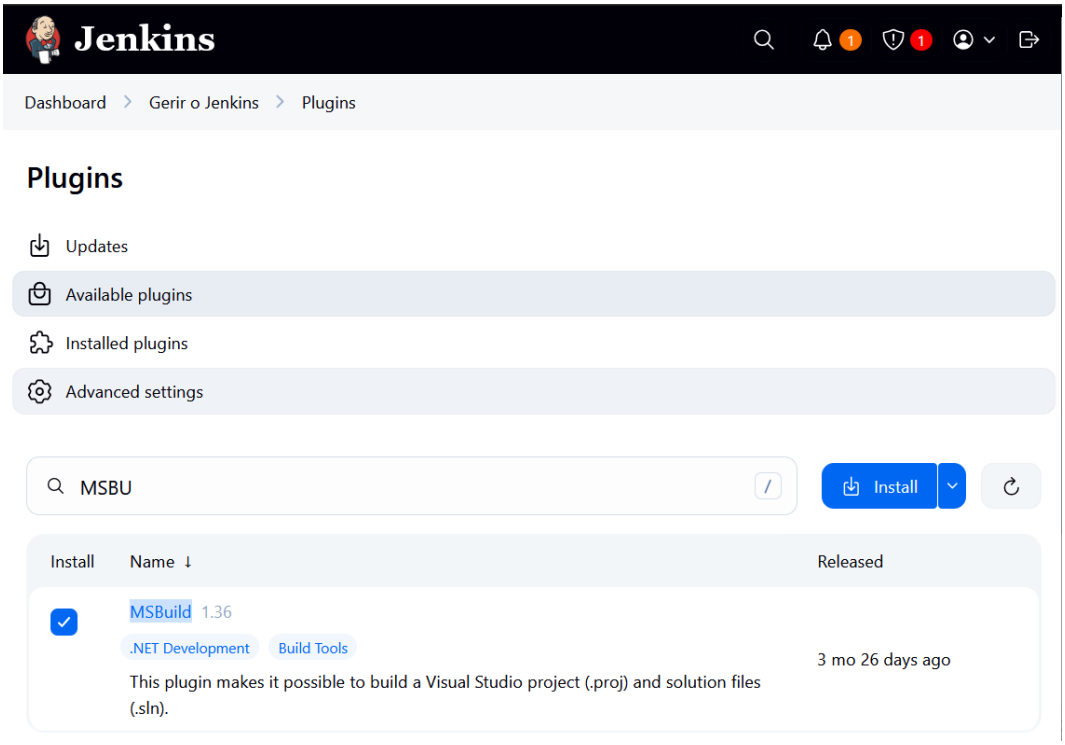

FIGURE 6.9: MSBuild Jenkins plugin

Figure 6.10 shows how to configure this plugin. To do this, first we move to the tools page on the Jenkins configuration:

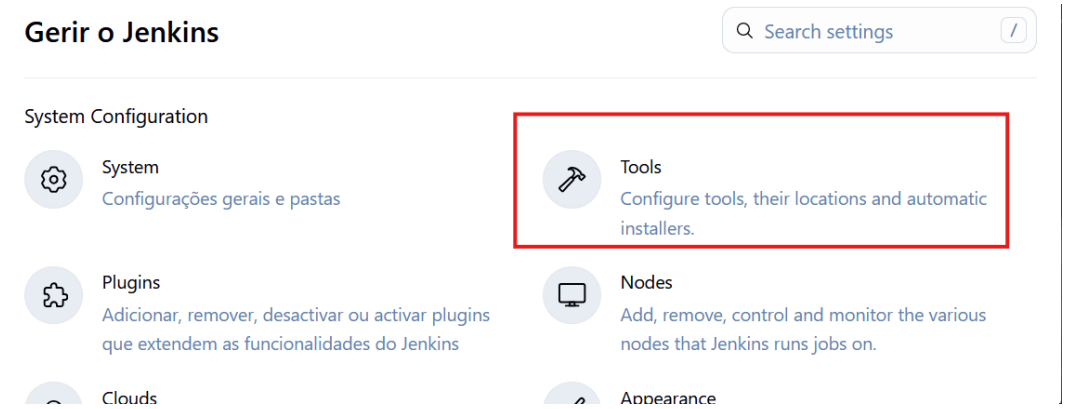

FIGURE 6.10: Jenkins tool page

After this step, the wanted MSBuild version was added as shown in the Figure 6.11.

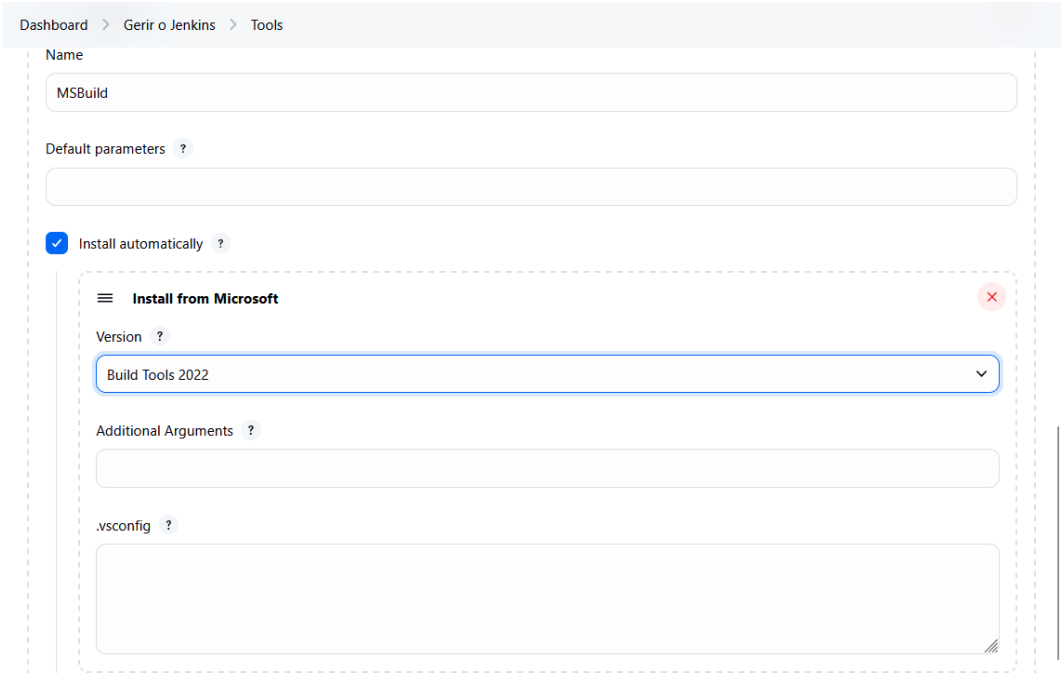

FIGURE 6.11: Jenkins MSBuild installer

Considering this tool, the initial Jenkinsfile created is represented in the Listing 6.7:

```
pipeline {
  agent any
  tools {
    msbuild 'MSBuild'
  }
  stages {
    stage('Build') {
      steps {
        script {
          bat "msbuild  /t:Rebuild"
        }
      }
    }
  }
}
```

LISTING 6.7: Jenkinsfile Build stage

6.2.6 Test

This subsection will explain the steps needed to comply with the functional requirement FR4.

The Test stage is also very similar to the Build stage in the sense that it is not possible to use .NET CLI and it needs to use another exe util. In order to Test .NET Framework applications the VSTest exe util needs to be used. Just like The MSBuild, there is a plugin for this on Jenkins called VSTest Runner.

Figure 6.12 shows how to get the plugin:

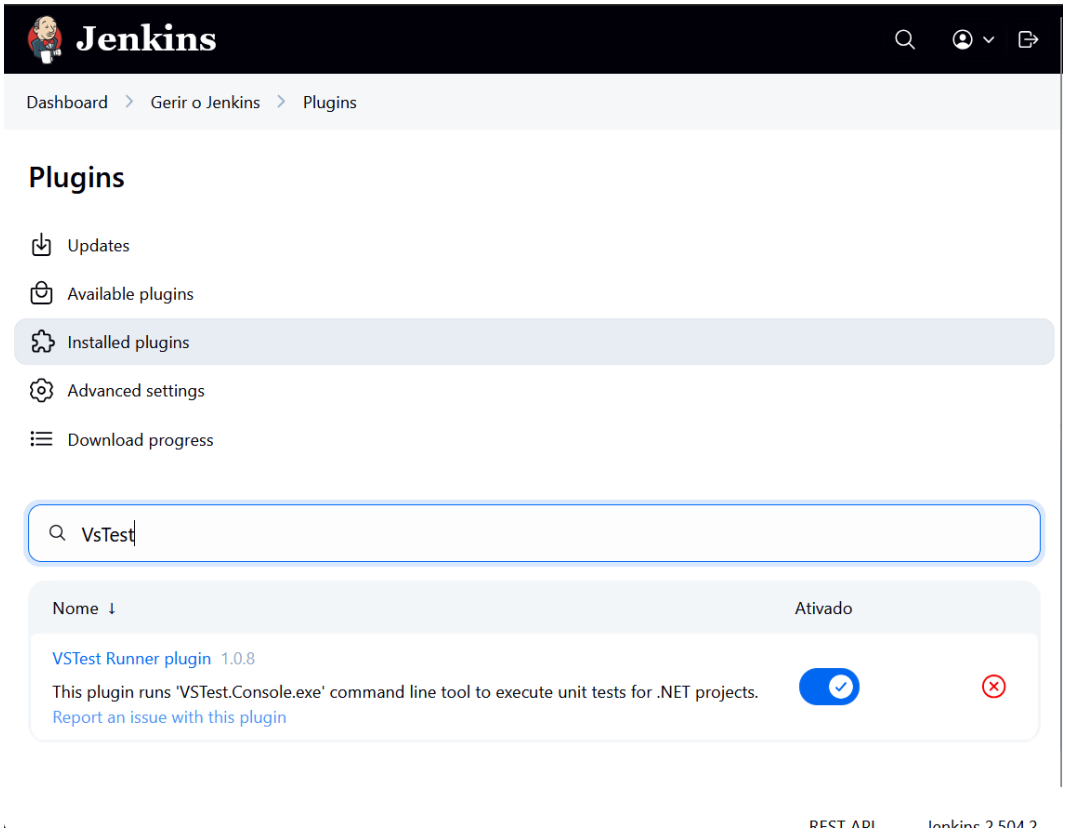

FIGURE 6.12: Jenkins VSTest tool

Figure 6.13 shows the configuration needed to use the VSTest tool:

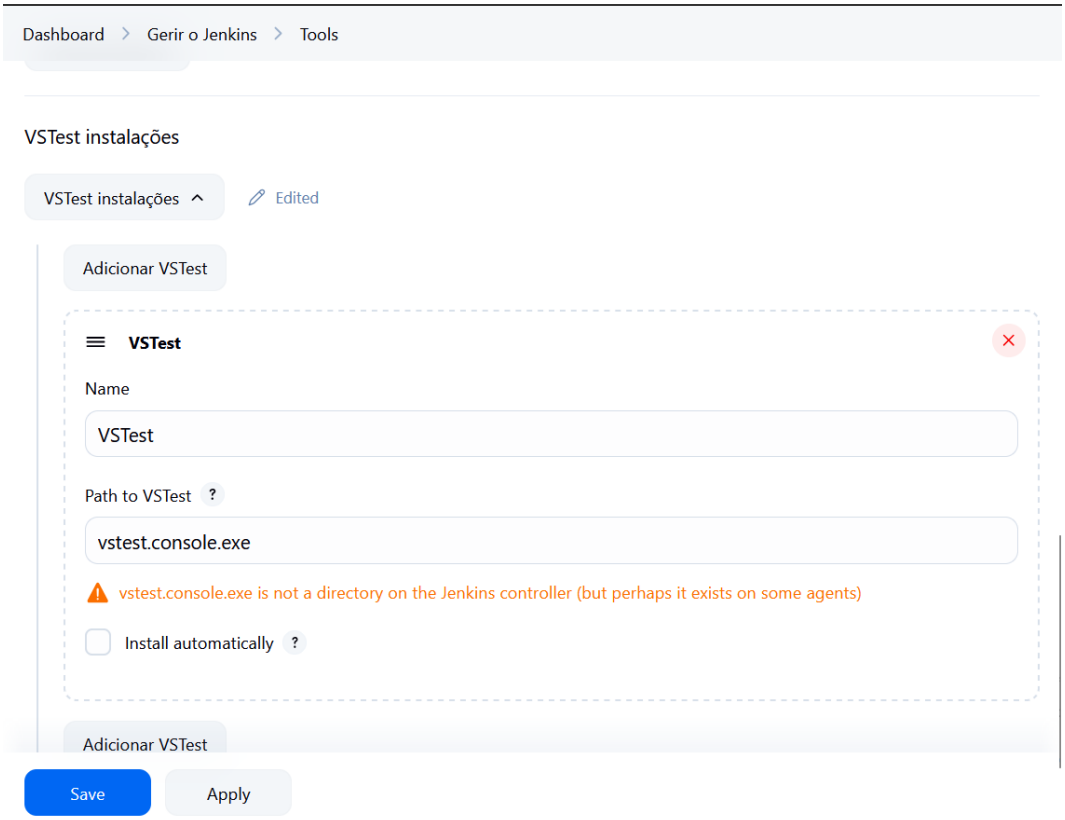

FIGURE 6.13: Jenkins VSTest configuration

Listing 6.8 shows the Jenkinsfile that includes both the Build and Test stages:

```
pipeline {
  agent any
  tools {
    msbuild 'MSBuild'
  }
  stages {
    stage('Build') {
      steps {
        script {
          bat "msbuild  /t:Rebuild"
        }
      }
    }
    stage('Test') {
      steps {
        dir('path to DLL') {
          echo "Starting to run the tests of the project."
          step([$class: 'VsTestBuilder',
              testFiles: 'ServicesTests.dll',
              vsTestName: 'VSTest',
              testCaseFilter: ''])
        }
      }
    }
  }
}
```

LISTING 6.8: Jenkinsfile Build and Test stages

6.2.7 Sonar

This subsection will explain the steps needed to comply with the functional requirement FR5.

This Sonar stage will allow static code analysis of the project. This is useful to understand possible code smells, vulnerabilities and to verify if the project is following certain quality gates.

In order to enable Sonarqube, a Docker container was created. Listing 6.9 shows the created docker compose file.

```
services:
  sonarqube:
    image: sonarqube:community
    hostname: sonarqube
    container_name: sonarqube
    read_only: true
    depends_on:
      db:
        condition: service_healthy
    environment:
      SONAR_JDBC_URL: jdbc:postgresql://db:5432/sonar
      SONAR_JDBC_USERNAME: sonar
      SONAR_JDBC_PASSWORD: sonar
    volumes:
      - sonarqube_data:/opt/sonarqube/data
      - sonarqube_extensions:/opt/sonarqube/extensions
      - sonarqube_logs:/opt/sonarqube/logs
      - sonarqube_temp:/opt/sonarqube/temp
    ports:
      - "9000:9000"
    networks:
      - ${NETWORK_TYPE:-ipv4}
  db:
    image: postgres:17
    healthcheck:
      test: ["CMD-SHELL", "pg_isready"]
      interval: 10s
      timeout: 5s
      retries: 5
    hostname: postgresql
    container_name: postgresql
    environment:
      PGUSER: sonar
      POSTGRES_USER: sonar
      POSTGRES_PASSWORD: sonar
      POSTGRES_DB: sonar
    volumes:
      - postgresql:/var/lib/postgresql
      - postgresql_data:/var/lib/postgresql/data
    networks:
      - ${NETWORK_TYPE:-ipv4}

volumes:
  sonarqube_data:
  sonarqube_temp:
  sonarqube_extensions:
  sonarqube_logs:
  postgresql:
  postgresql_data:

networks:
  ipv4:
    driver: bridge
    enable_ipv6: false
```

```
  dual:
    driver: bridge
    enable_ipv6: true
    ipam:
      config:
        - subnet: "192.168.2.0/24"
          gateway: "192.168.2.1"
        - subnet: "2001:db8:2::/64"
          gateway: "2001:db8:2::1"
```

LISTING 6.9: Docker compose file containing SonarQube and PostgreSQL database

In order to create the container, the command listed in the Listing 6.10 will have to be executed in the same root as the docker-compose file location.

```
docker-compose up
```

LISTING 6.10: Docker compose command

This file will create a Docker container with SonarQube and a PostgreSQL database inside. This database is used in order to enable SonarQube to persist the necessary data.

After this, you can access Sonar using the default port on localhost, so, `http://localhost:9000`.

Figure 6.14 shows how to create a new project inside SonarQube.

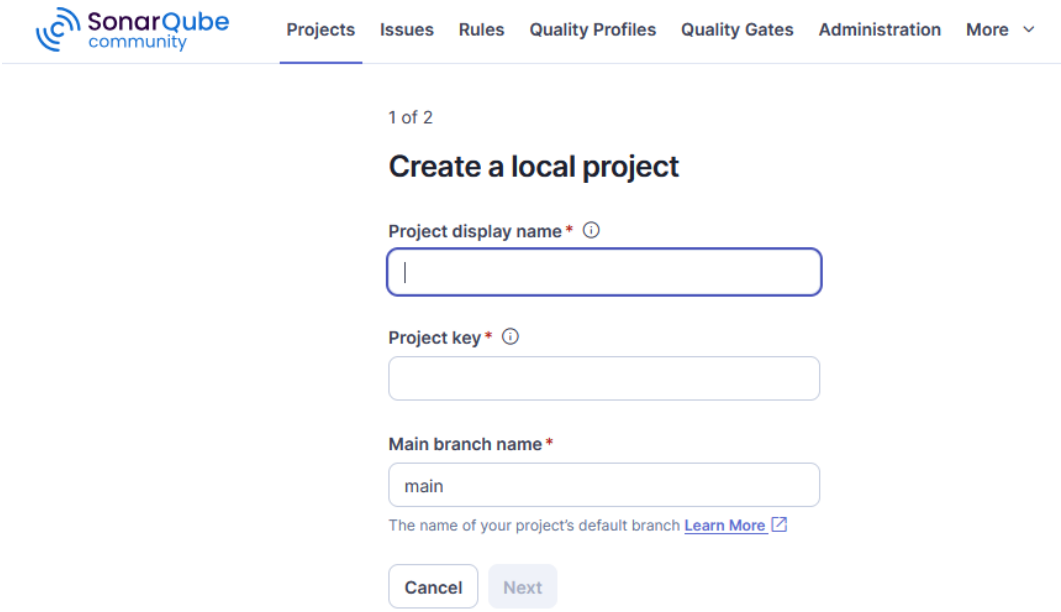

FIGURE 6.14: Sonar Project creation

After creating the project we will go to its page in SonarQube and select the option Other CI as shown in Figure 6.15.

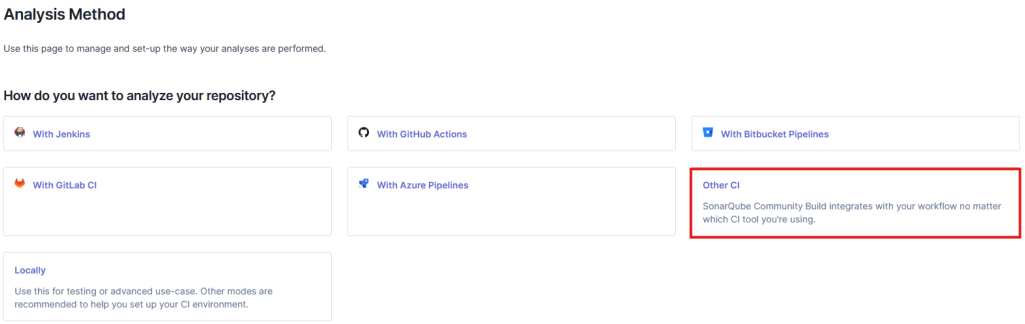

FIGURE 6.15: Sonar Analysis Method selection

Normally we would go for the "With Jenkins" option, but since we are using a Local Repo instead of a remote one, this option is the one to go with. After that, we will ask Sonar to provide a token and select the project dependencies, so in this specific scenario, it is .NET Framework as shown in Figure 6.16.

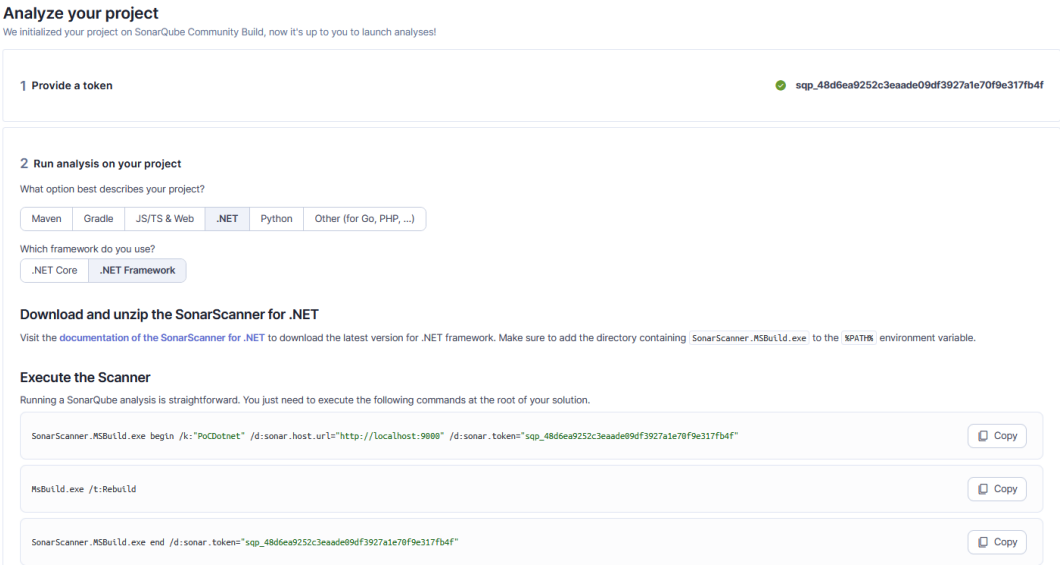

FIGURE 6.16: Sonar project settings selection

After configuring everything in Sonar, we still need to do one more thing in Jenkins. The plugin SonarQube Scanner should be installed as shown in the Figure 6.17.

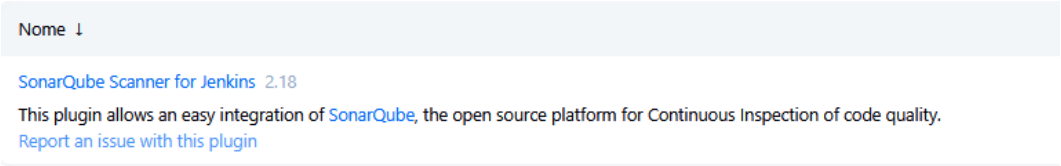

FIGURE 6.17: SonarQube Scanner plugin

It will also be needed to configure the plugin. Figure 6.18 shows that configuration present on the Tools tab of Jenkins.

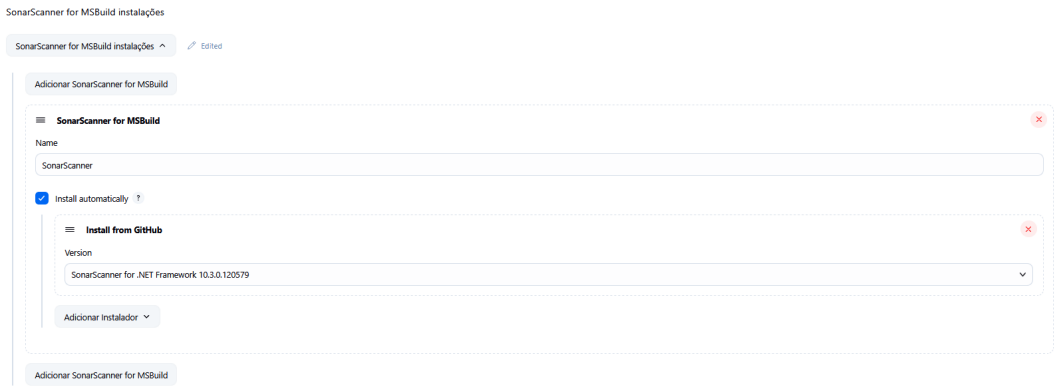

FIGURE 6.18: SonarQube Scanner configuration

Finally, the configuration shown in the Listing 6.11 is necessary in order to enable analysis of the project.

```
pipeline {
  agent any
  tools {
    msbuild 'MSBuild'
  }
  stages {
    stage('Build') {
      steps {
        script {
          bat "msbuild  /t:Rebuild"
        }
      }
    }
    stage('SonarQube Begin Analysis and Project Compilation') {
      steps {
        script {
          def scannerHome = tool 'SonarScanner'
          def msbuildHome = tool 'MSBuild'
          withSonarQubeEnv() {
            bat "\"${scannerHome}\\SonarScanner.MSBuild.exe\" begin /k:\"
    PoCDotnet\" /d:sonar.host.url="http://localhost:9000" /d:sonar.token="
    sqp_48d6ea9252c3eaade09df3927a1e70f9e317fb4f""
            bat "\"${msbuildHome}\\MSBuild.exe\" /t:Rebuild"
            bat "\"${scannerHome}\\SonarScanner.MSBuild.exe\" end"
          }
        }
      }
    }
    stage('Test') {
      steps {
        dir('path to DLL') {
          echo "Starting to run the tests of the project."
          step([$class: 'VsTestBuilder',
              testFiles: 'ServicesTests.dll',
              vsTestName: 'VSTest',
              testCaseFilter: ''])
        }
      }
    }
  }
```

```
39 }
```

Listing 6.11: Jenkinsfile Build and Test stages

Figure 6.19 shows the results of the analysis.

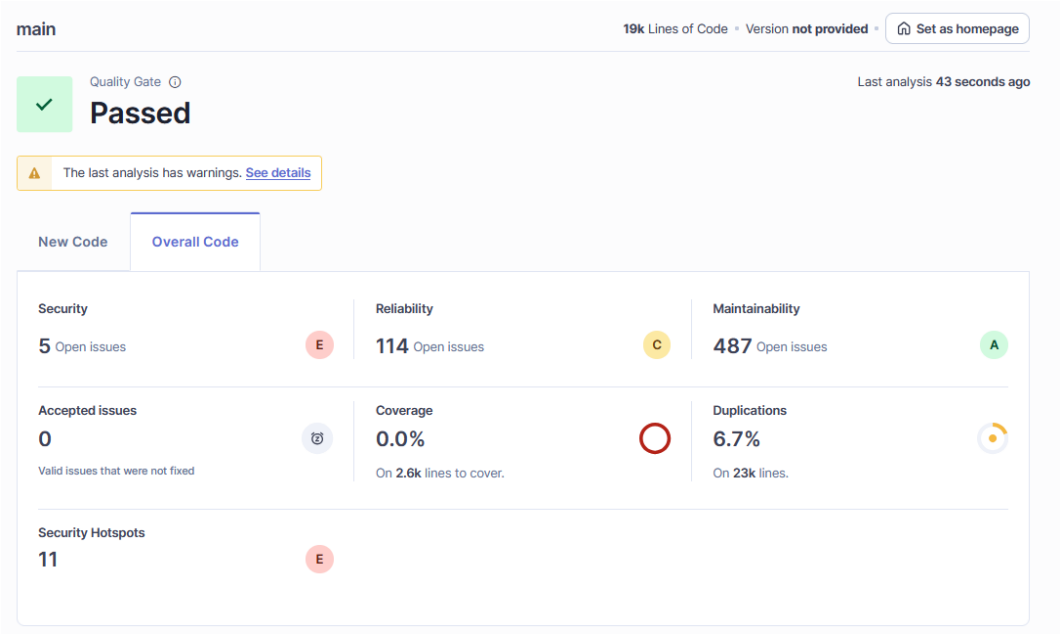

Figure 6.19: SonarQube Main branch analysis

Currently the coverage is zero, because the Test DLLs were not available to run, because of the lack of some project dependencies, which made the code coverage unavailable. In a normal scenario, this would not happen.

6.3 Results

A number of useful outcomes from the CI/CD pipeline implementation for the DOMUS project show the potential and challenges of integrating automated procedures into a sophisticated legacy system. Build, test, and static code analysis—the three main steps outlined in the functional requirements—were successfully automated by the pipeline, offering a methodical and repeatable software delivery process.

By using its dedicated plugin, the build stage verified that it was possible to integrate MSBuild with Jenkins and compile.NET Framework 4 projects without requiring a complete Visual Studio installation. Given the limitations imposed by the DOMUS ecosystem, this result is especially noteworthy because it confirmed that current dependencies could be reliably resolved within an automated pipeline.

Additionally, the test stage—powered by the VSTest Jenkins plugin—was successfully configured and run, allowing project DLLs to be automatically validated. This made it possible to perform regression testing without the need for human intervention, increasing reliability and lowering the possibility of missed flaws.

By incorporating static code analysis into the pipeline, SonarQube's integration added even more value. Reports pointing out potential vulnerabilities, code smells, and maintainability issues were generated by the analysis. These observations are a significant step in enhancing the codebase's long-term quality and sustainability, even though they are not immediately necessary for the system's successful development or deployment.

However, it should be noted that the pipeline's deployment stage could not be implemented due to time constraints.

6.3.1 CI/CD Maturity Model Evaluation

This subsection compares the pipeline's implementation to the Continuous Delivery 3.0 Maturity Model (CD3M), which was presented in Section 3.1.6. The CD3M divides maturity into five levels: Foundation, Novice, Intermediate, Advanced, and Expert. It also divides maturity into five categories: Continuous Intelligence, Continuous Planning, Continuous Integration, Continuous Testing and Continuous Deployment.

Continuous Intelligence and Continous Planning

For this two categories, it is not really possible to evaluate their level, since there was no access provided to the backlog or how are business decisions made based on gathered data. This two categories will also be heavily dependent on what DOMUS module is being analyzed, so it is not really valid for the study being developed.

Continous Integration

The pipeline exhibits characteristics in Continuous Integration that fall between Novice and the lower bound of Intermediate. By using the specialized Jenkins tooling to automate builds for .NET Framework 4 components using MSBuild, a complete Visual Studio installation is not required, and compilation is guaranteed to be repeatable also on the CI worker, including running during night, outside working hours since it is fully automatic. Jenkins' multibranch discovery and execution, in conjunction with this setup, offers reliable integration health feedback for each branch or pull request. The pipeline also includes static code analysis, that also provides better code reliability and improves code quality. Even though the evidence is strong, its main focus is still on creating trustworthy compilations rather than implementing more complex integration practices that are indicative of an obvious Intermediate placement, like generating reports or triggering builds after a commit of a new feature. Continuous Integration is therefore rated as mid level Novice.

Continuous Testing

Continuous Testing involves the execution of automated tests via VSTest within the pipeline during each run, facilitating the detection of regressions without the need for manual intervention. This represents a significant advancement beyond the Foundation level and meets Novice standards, as it incorporates automated unit

and basic integration tests into the pipeline. As a result, Continuous Testing is categorized as Novice.

Continuous Deployment

In Continuous Deployment, the maturity level is classified as Foundation. The architectural design segregated deployment into a dedicated Jenkins job to maintain distinct verification and release processes, thereby adhering to the established release governance in DOMUS. The deployment stage was not executed within the dissertation's timeframe, indicating that the transition from a validated build to an operational release continues to be a manual process. The absence of an automated, parameterized, and environment-aware deployment path is the primary reason for the overall maturity being at the Foundation level instead of Novice.

Figure 6.20 shows the result analyzed using the Continous Delivery 3.0 Maturity Model.

	FOUNDATION Developing on a CD 3.0 platform, but the cycle is poorly automated.	**NOVICE** Working with basic automation on a reactive level.	**INTERMEDIATE** Running average CD 3.0 technologies with proactive elements.	**ADVANCED** Working with advanced CD 3.0 tech, that is quantitatively managed.	**EXPERT** Increasingly utilizing AI to improve the CD 3.0 development cycle.
INTELLIGENCE Making business decisions based on gathered used data.	Tracking customer behaviour on a server and receive feedback.	Basic monitoring of app usage and handling customer feedback	Advanced customer monitoring and A/B testing in place.	Receiving predefined metrics and reports. Decisions being made based on detailed analytics.	Real-time data collection analysis and reporting using AI
PLANNING Automating backlog item creation and prioritization to improve collaboration.	Managing the complete backlog on a centralized server.	Managing all work by means of a digital backlog.	Automatically creating items for the backlog.	Automatically receiving backlog prioritization suggestions.	Automatically prioritizing and creating backlog items using AI.
INTEGRATION Automatically building your software to shorten the development cycle.	Running a centralized version control system. Running a centralized build server	Running a workflow orchestrator and receiving reports. Running builds while the company is asleep.	Triggering builds after the commit of a new feature.	Running integrations on a scalable microservice architecture. Staged Integrations - compiling the source code that was edited only	Automatically scaling continuous integration services.
TESTING Automatically testing newly developed features to avoid tedious work.	Running a centralized unit-test server. Manually starting unit tests.	Running unit tests in a Continuous Delivery pipeline. Manually starting your automated integration tests.	Triggering integration tests in your Continuous Delivery pipeline. Manually starting your automated acceptance-tests.	Triggering acceptance tests in your Continuous Delivery pipeline. Manually starting your automated security and performance tests.	Triggering end-2-end regression tests in your pipeline.
DEPLOYMENT Automatically deploying new builds to scalable environments.	Running a centralized deployment server.	Running basic deployment scripts. Automatically deploying to a test server after a successful build.	Automatically deploying to the production server using a pipeline.	Automatically deploying without any downtime.	Automatically deploying on endless scalable platforms.

FIGURE 6.20: Continuous Delivery 3.0 Maturity Model results.

All things considered, the pipeline satisfied the primary functional needs of static analysis, regression testing, automation, and compilation verification. The outcomes demonstrate that a CI/CD process can be successfully implemented in the DOMUS

ecosystem, leading to quantifiable enhancements in quality assurance and process repeatability.

6.4 Summary

This chapter presented the experimentation process carried out to design and implement a CI/CD pipeline tailored to the DOMUS ecosystem.

Jenkins was selected as the main automation server in the technological stack, which also included SonarQube for static code analysis, MSBuild for compilation, VSTest for automated testing, and Git for version control. Although Docker was first contemplated for containerized execution, the final implementation depended on Jenkins' WAR distribution because of the limitations of working with local repositories.

The project's limitations and presumptions were brought to light by the experimentation, especially the reliance on Windows for .NET Framework 4 compatibility and the implementation's restricted scope to a single repository. Notwithstanding these limitations, the pipeline effectively integrated static code analysis and automated the build and test phases, offering a dependable and repeatable software delivery workflow.

The outcomes showed observable increases in process automation, quality assurance, and reliability as well as validated the viability of implementing CI/CD practices to DOMUS. However, time constraints prevented the deployment stage from being finished, so this remains a potential area for future research. As a result, the experiment confirmed the primary functional requirements while also highlighting issues with environment configuration and dependency management.

Chapter 7

Conclusion

In the overall setting of DOMUS, a sizable information system at P.PORTO that has not traditionally used automated development and deployment procedures, this dissertation investigated the deployment and assessment of a CI/CD pipeline. In accordance with the Design Science Research methodology and PRISMA methodology, the work concentrated on developing a working proof of concept that addressed significant inefficiencies in quality assurance, testing, and versioning.

Despite its technical limitations due to its dependence on .NET Framework 4 and the use of local repositories, the implementation process effectively illustrated the viability of applying CI/CD to DOMUS. Time constraints prevented the pipeline from being fully completed, since the deployment step is missing, but the build, test, and static code analysis phases significantly increased automation and reliability, confirming the main goals of the study.

7.1 Research Question

How can DevOps contribute to mitigate the inefficiencies of the current DOMUS lifecycle?

By automating repetitive tasks, lowering manual errors, and introducing continuous validation mechanisms, the experimentation demonstrated that DevOps practices, when implemented through CI/CD pipelines, directly mitigate inefficiencies. Jenkins's integration of MSBuild and VSTest made it possible for compilation and regression testing to be done automatically, and SonarQube's introduction of systematic static code analysis helped to identify problems that might otherwise go overlooked.

Even though deployment was not accomplished, the partial implementation had already shown quantifiable improvements in quality assurance and process repeatability. The lack of automated deployment and ongoing monitoring has prevented full maturity from being achieved, despite the presence of basic automation for building, testing, and code quality analysis.

According to the CD3M appraisal, this results indicate a maturity profile with Continuous Integration at mid-Novice, Continuous Testing at Novice, and Continuous Deployment at Foundation. This explains the observed improvements in compilation reliability, regression detection, and static analysis, while lead-time to release and operational feedback loops have not shown similar progress. DevOps effectively

reduced inefficiencies primarily in areas where automation was implemented, such as build processes, testing, and code quality. The evaluation clarifies the sources of observed gains and identifies remaining bottlenecks: advancing toward an environment-aware and parameterized deployment job would enhance the pipeline, while linking delivery events with planning data and telemetry would facilitate learning across iterations.

7.2 Contributions

This dissertation makes both academic and practical contributions. In practice, it offers a roadmap for expanding automation across DOMUS's numerous repositories and a verified proof of concept that shows how CI/CD pipelines can be incorporated into a legacy system. Additionally, it provides an example Jenkins configuration that incorporates tooling specific to .NET Framework, which is frequently difficult in contemporary DevOps environments.

Academically, this dissertation advances knowledge about the adoption of DevOps in institutionally limited settings, especially when legacy technologies impose stringent constraints. It also demonstrates how theoretical evaluation frameworks, like the CI/CD maturity model, can be used to evaluate incremental progress even in the absence of a fully functional pipeline.

7.3 Limitations

The deployment stage, which was not finished because of time constraints, is the main limitation of this work. Because of this, the pipeline was unable to be assessed comprehensively, which left open concerns regarding how well it would function in real-world settings. The experimentation was also limited to a single DOMUS repository rather than the entire ecosystem, which is another drawback.

Furthermore, containerized approaches were not as feasible due to their dependence on local repositories, which prevented a fully portable solution and limited experimentation to Jenkins WAR execution. Although these restrictions limited the results' generalizability, they still offer a solid basis for further development.

It's also critical to remember that expanding this implementation to the full DOMUS system would necessitate a substantial increase in human resources, which are currently unavailable. Given the size and complexity of the system, this lack of resources may be one of the primary obstacles to overcoming some of the limitations noted.

7.4 Risk Assessment

Considering the table A.1 we can also confirm that some of the risks that were analyzed happened:

- **ID 1**: As explained in the section 7.3, there were time constraints which made it so that the CI/CD implementation was not finalized.

- **ID 2**: This risk happened not because of a lack of knowledge of DORA metrics, but because of the lack of deployment implementation.
- **ID 3**: This ID had one of the highest PI scores, but it was avoided as much as possible, which made it a non-issue most of the time, since there was a proper analysis of the DOMUS infrastructure before the implementation of the CI/CD pipeline.
- **ID 4**: Just like ID 2, this risk happened because of the lack of deployment implementation.
- **ID 5**: Just like ID 3, this one also had one of the highest PI scores, but just like it, it was also avoided as much as possible, since the DOMUS infrastructure limitations and incompatibilities were studied before the implementation process.
- **ID 6**: This risk was one of the main reasons why the implementation was not fully finished. The main cause why the deployment was not finished like explained in the section 7.3.
- **ID 7**: This risk also affected the dissertation, since the lack of the deployment, made it so that the conclusion was not as good as was expected to be since there is no DORA metrics analysis.

The risk assessment analysis leads to the conclusion that, despite its limitations in accurately predicting every possible risk, it is an essential tool for preventing and reducing them. In this particular instance, it forced a careful analysis of the risks and the DOMUS infrastructure, which decreased the possibility of future mistakes and possible time losses.

7.5 Future Work

In order to ensure end-to-end automation from code integration to production delivery, future work should first concentrate on finishing the pipeline's deployment stage. This will enable a thorough assessment using DORA metrics, specifically the time to restore services and the frequency of deployments.

By extending the proof of concept to more DOMUS repositories, scalability would be confirmed and any necessary repository-specific modifications would be highlighted. Prioritizing the introduction of Docker and Docker Compose is also necessary because containerization would greatly improve portability, reproducibility, and maintenance ease.

To further align DOMUS practices with advanced DevOps maturity levels and enhance system health visibility, continuous monitoring and feedback loops could be incorporated into the pipeline.

7.6 Personal Remarks

This dissertation was both technically difficult and personally fulfilling to develop. The DOMUS ecosystem's limitations necessitated innovative approaches to problem-solving and reaffirmed the value of flexibility in practical software engineering. The experience demonstrated the transformative power of DevOps practices, even when implemented gradually.

The student's proficiency with CI/CD tools, testing frameworks, and quality assurance procedures was further enhanced by this work, which also constituted a substantial learning experience. Outside of academic context, the knowledge acquired will influence professional practice in upcoming projects where efficiency, automation, and dependability are still crucial.

Bibliography

[1] *DOMUS — Escola Superior de Saúde*. URL: https://www.ess.ipp.pt/informacoes/centro-de-informatica/domus.

[2] Jan vom Brocke, Alan Hevner, and Alexander Maedche. "Introduction to Design Science Research". In: Sept. 2020, pp. 1–13. ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

[3] Alberto Borges Pereira. "Planeamento da Dissertação Riscos em projetos". In: (). ISSN: 2024/2025.

[4] Ramtin Jabbari et al. "What is DevOps? A Systematic Mapping Study on Definitions and Practices". In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. XP '16 Workshops. Edinburgh, Scotland, UK: Association for Computing Machinery, 2016. ISBN: 9781450341349. DOI: 10.1145/2962695.2962707. URL: https://doi.org/10.1145/2962695.2962707.

[5] Liming Zhu, Len Bass, and George Champlin-Scharff. "DevOps and Its Practices". In: *IEEE Software* 33.3 (2016), pp. 32–34. DOI: 10.1109/MS.2016.81.

[6] *What is CI/CD?* URL: https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[7] *What Are CI/CD And The CI/CD Pipeline? | IBM*. URL: https://www.ibm.com/think/topics/ci-cd-pipeline.

[8] *What Is a Continuous Delivery Maturity Model (CDMM)? | Codefresh*. URL: https://codefresh.io/learn/continuous-delivery/what-is-a-continuous-delivery-maturity-model-cdmm/.

[9] *Continuous Delivery 3.0 Maturity Model — NISI - Nederlands Instituut voor de Software Industrie*. URL: https://nisi.nl/continuousdelivery/articles/maturity-model.

[10] *DORA Metrics: How to measure Open DevOps Success | Atlassian*. URL: https://www.atlassian.com/devops/frameworks/dora-metrics.

[11] *Use Four Keys metrics like change failure rate to measure your DevOps performance | Google Cloud Blog*. URL: https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance.

[12] *Team Tools - The State of Developer Ecosystem in 2023 Infographic | JetBrains: Developer Tools for Professionals and Teams*. URL: https://www.jetbrains.com/lp/devecosystem-2023/team-tools/#tools_cloud.

[13] Sai Teja Makani and Shivadutt Jangampeta. "The Evolution of CICD Tools In Devops from Jenkins to Github Actions". In: *International Journal of Computer Engineering and Technology (IJCET)* 13 (2), pp. 166–174.

[14] *Jenkins User Documentation*. URL: https://www.jenkins.io/doc/.

[15] *Understanding GitHub Actions - GitHub Docs*. URL: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions.

[16] *Continuous Integration and Delivery | GitLab*. URL: https://about.gitlab.com/solutions/continuous-integration/.

[17] *What is Azure Pipelines? - Azure Pipelines | Microsoft Learn*. URL: https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops.

[18] *Get started with Bitbucket Pipelines | Bitbucket Cloud | Atlassian Support*. URL: https://support.atlassian.com/bitbucket-cloud/docs/get-started-with-bitbucket-pipelines/.

[19] *What is .NET? An open-source developer platform | .NET*. URL: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet.

[20] *What is .NET Framework? A software development framework | .NET*. URL: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework.

[21] *What is Docker? | Docker Docs*. URL: https://docs.docker.com/get-started/docker-overview/.

[22] *SonarQube Server 2025.3 | Documentation*. URL: https://docs.sonarsource.com/sonarqube-server/latest/.

[23] Marco Kuhrmann, Daniel Méndez Fernández, and Maya Daneva. "On the Pragmatic Design of Literature Studies in Software Engineering: An Experience-based Guideline". In: *Empirical Software Engineering* 22 (Dec. 2017), pp. 2852–2891. DOI: 10.1007/s10664-016-9492-y.

[24] Matthew J. Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021), n71. DOI: 10.1136/bmj.n71. URL: https://www.bmj.com/content/372/bmj.n71.

[25] Matthew J. Page et al. "Updating guidance for reporting systematic reviews: development of the PRISMA 2020 statement". In: *Journal of Clinical Epidemiology* 134 (2021), pp. 103–112. DOI: 10.1016/j.jclinepi.2021.02.003. URL: https://doi.org/10.1016/j.jclinepi.2021.02.003.

[26] Kati Kuusinen et al. "A Large Agile Organization on Its Journey Towards DevOps". In: *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2018, pp. 60–63. DOI: 10.1109/SEAA.2018.00019.

[27] Holger Nehls and Daniel Ratiu. "Towards Continuous Delivery for Domain Experts: Using MDE to Integrate Non-Programmers into a Software Delivery Pipeline". In: *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. 2019, pp. 598–604. DOI: 10.1109/MODELS-C.2019.00091.

[28] Suha Afaneh et al. "Security Challenges Review in Agile and DevOps Practices". In: *2023 International Conference on Information Technology (ICIT)*. 2023, pp. 102–107. DOI: 10.1109/ICIT58056.2023.10226018.

[29] Hasan Hameed Hasan Ahmed Abdulla and Fawzi Abdulaziz Albalooshi. "Automated Testing for DevOps in GitHub Environment: A Comprehensive Analysis". In: *2023 International Conference on Advanced Mechatronics, Intelligent Manufacture and Industrial Automation (ICAMIMIA)*. 2023, pp. 833–838. DOI: 10.1109/ICAMIMIA60881.2023.10427702.

[30] M.A.W. Karunarathne, W. M. J. I. Wijayanayake, and A. P. K. J. Prasadika. "DevOps Adoption in Software Development Organizations: A Systematic Literature Review". In: *2024 4th International Conference on Advanced Research in Computing (ICARC)*. 2024, pp. 282–287. DOI: 10.1109/ICARC61713.2024.10499789.

Appendix A

Risk Assessment

Table A.1: Risk Assessment

ID	Description	Cause	Effect	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Response Type	Response Description
1	Delay in implementing CI/CD pipeline	Complexity of integration	Project timeline extended	3	4	12	Inability to meet dissertation deadlines	Mitigate	Break down tasks, set milestone reviews, allocate extra resources.
2	Inadequate expertise in DORA metrics analysis and CI/CD Maturity model	Limited practical experience	Misinterpretation of pipeline performance	2	3	6	Inaccurate evaluation of CI/CD effectiveness	Mitigate	Gain hands-on experience through case studies and online tools.
3	Lack of compatibility with DOMUS infrastructure	Technical constraints	CI/CD implementation fails	3	5	15	Project goals not achieved	Avoid	Conduct a feasibility analysis and select compatible tools.

ID	Description	Cause	Effect	Probability (1-5)	Impact (1-5)	PI Score	Expected Result, No Action	Response Type	Response Description
4	Errors in deployment automation scripts	Lack of testing or human error	Deployment process fails	3	4	12	Delays in deployment and increased debugging time	Mitigate	Implement comprehensive testing for scripts before production.
5	Incompati-bility of selected CI/CD tools	Outdated infrastructure	Tools fail to integrate with DOMUS	4	4	16	Pipeline implementation fails	Avoid	Choose tools after conducting compatibility assessments.
6	Time constraints for pipeline evaluation	Overlapping academic and project deadlines	Incomplete evaluation	4	3	12	Reduced reliability of results	Accept	Allocate dedicated evaluation time in project schedule.
7	Lack of Clear Conclusion or Findings	Insufficient data analysis.	Weak final report with limited insights or actionable recommenda-tions.	2	5	10	Risk of weak project outcomes and limited academic value.	Accept	Acknowledge constraints and focus on presenting actionable findings based on available data.